Universitat de Lleida

TREBALL FINAL DE GRAU

Estudiant: Ricard Cervera Enrich

Titulació: Grau en Enginyeria Informàtica

Títol de Treball Final de Grau: **Desenvolupament d'un videojoc 3D shooter i plataformes**

Director/a: **Francesc Sebe Feixas**

Presentació

Mes: Setembre

Any: 2019

Índex

Índex de figures

Capítol 1. Introducció

El Treball de Fi de Grau (TFG) ha tingut com a objectiu el desenvolupament d'un videojoc que combinés *shooter* i plataformes. En aquest document s'explica quines decisions de disseny s'han pres per a obtenir una bona base i així poder acabar amb èxit el desenvolupament.

El videojoc és un tipus de projecte molt ambiciós, ja que per aconseguir acabar-lo amb èxit, són necessaris professionals d'àmbits molt diferents com la informàtica, l'art, la música, el màrqueting, etc.

Per a aquest projecte he utilitzat principalment dos programes, Unity i Blender.

Tot i l'ambiciós que era el desenvolupament d'aquest videojoc i la limitació de temps de la que disposava, volia poder arribar a presentar un mínim del videojoc i que es pogués arribar a jugar en l'entrega del TFG. Per aquest motiu, no vaig seguir l'ordre usual a l'hora de desenvolupar videojocs, sinó que vaig començar per explorar els àmbits del projecte que no coneixia tan bé. Pel que fa a Unity, ja tenia una petita base i, per aquest motiu vaig preferir començar pels models 3D, ja que aquesta feina no semblava senzilla. Així doncs, vaig decidir començar per Blender, ja que no l'havia utilitzat mai i em semblava interessant fer el meu propi personatge pel treball. Vaig aprendre a modelar, a dissenyar l'estructura de nodes que mourien al personatge i, finalment, a crear animacions. Un cop adquirits els coneixements bàsics, vaig poder fer estimacions sobre el temps que tardava en crear animacions, el que em va permetre deixar la resta per més endavant. Degut a la inexperiència, vaig tenir que retornar a Blender per tal de millorar tant el model com el *rig*, ja que el disseny inicial no em permetia fer algunes animacions més complexes. Ja a Unity, vaig començar per les físiques, la part més bàsica de qualsevol videojoc, i després la càmera (una versió més simplificada de l'actual). En aquest punt, per acabar el personatge ja només em faltava el disseny del sistema d'armes i les mateixes armes. Després de crear alguns enemics i objectes clau del videojoc, com podrien ser cristalls, caixes i munició, vaig posar-me a treballar en els menús i, finalment, en el disseny dels mapes.

1.1 Objectius

El principal objectiu d'aquest treball era crear un videojoc divertit i fàcil de jugar, i que, al mateix temps, tingués una història entretinguda. Per aconseguir-ho, s'han implementat uns controls àgils que permetin al jugador realitzar múltiples accions simultàniament (com per exemple, saltar i disparar al mateix moment), així com una àmplia varietat d'armes i objectius en els que utilitzar-les. De la mateixa manera, s'implementaria un sistema de nivells que animés i encuriosís al jugador a utilitzar al màxim la varietat d'armes disponibles (Figura 1).

Un altre objectiu era la millora de l'ús de programes pertanyents a l'entorn del videojoc com Unity i Blender. En relació al primer, tot i que ja l'havia utilitzat prèviament, volia aprofundir molt més i aprendre noves tècniques per tal d'augmentar la qualitat del joc; respecte al segon, no només tenia la intenció d'aprendre com funcionava i d'adquirir els coneixements bàsics, sinó que pretenia ser capaç de realitzar tot el procés de creació d'un personatge.

1.2 Descripció del joc

El videojoc té lloc en un futur pròxim, en un univers on conviuen milions d'espècies. Acompanyarem en la seva aventura al nostre protagonista, un noi jove no hominoide que viu dins el clavegueram sota una gran ciutat industrialitzada. Aquí és on comença tot, el seu amic desapareix, i aquest comença a sospitar que alguna cosa li ha passat. Buscant-lo, explora tot el clavegueram de la ciutat. Sense sort, decidirà pujar a la superfície però, de sobte, es troba en una situació hostil on s'haurà de fer pas a través de nombrosos enemics. Enmig de la batalla podrà veure com introdueixen al seu amic en una nau espacial i se l'enduen. El nostre protagonista els perseguirà per tota la galàxia fins a recuperar el seu gran amic...

Figura 1: Imatge videojoc finalitzat.

1.3 Interfície

Aquest videojoc es jugarà des d'un ordinador i seran necessaris un teclat i un ratolí.

Ratolí:

- Amb el moviment del ratolí es mourà la càmera que segueix al personatge.
- Fent clic al botó dret, atacarem amb l'arma secundaria (cos a cos); si cliquem múltiples vegades seguides, realitzarem una combinació d'atacs cos a cos.
- Utilitzant el botó esquerre del ratolí, dispararem l'arma equipada i, si no en tenim cap d'equipada, equiparà l'ultima utilitzada.

Teclat:

- Amb les tecles "W", "S", "D" i "A" es girarà i es mourà el personatge, endavant, endarrere, dreta i esquerra respectivament.
- Amb la combinació d'una de les tecles anteriorment mencionades ("W", "S", "D" o "A") i la tecla "Mayús" es mourà al personatge en diferents direccions, però fixant la vista endavant, és a dir, sense fer que aquest es giri en la direcció en la que es mou.
- Amb la barra espaiadora, el personatge saltarà. Així mateix, es pot prémer dues vegades per a fer un doble salt.
- La tecla "Q" ens obrirà el menú per a poder seleccionar armes.
- Utilitzant la tecla "E" es podrà interaccionar amb els elements del joc (botigues, portes, etc.)
- Amb la tecla "Escape", s'obrirà el menú principal.

1.4 Plataformes utilitzades

1.4.1 Unity

Unity és un motor gratuït per a desenvolupar videojocs amb C#. Disposa d'una gran comunitat, el que facilita molt a principiants començar a crear els seus propis videojocs. Amb Unity es poden crear tant videojocs 2D com 3D i aquests es poden compilar per a quasi totes les plataformes disponibles al mercat.

Unity disposa d'una interfície gràfica molt útil que ajuda a posicionar objectes al mapa, afegir components i *scripts* mitjançant el sistema de "agafa i arrastra". Aquest sistema fa molt ràpid i intuïtiu el desenvolupament amb aquesta eina.

Tot seguit, s'explicaran uns quants components de Unity que ajudaran a entendre la documentació que ve a continuació:

Escena

Per aconseguir un millor rendiment, els videojocs sempre s'han separat per nivells. D'aquest manera, no hem de carregar tot el joc de cop ni hem de tenir recursos ocupats si no els hem d'utilizar. Dins de Unity podem dividir en escenes tant els nivells com el menús.

GameObject

Els *GameObjects* són qualsevol del objectes que podem trobar dins l'escena, i poden contenir components de Unity, *scripts*, panels, etc. Per a poder accedir a aquests *scripts* o components des de dins de codi utilitzarem el mètode "GameObject.getComponent<Tipus de component>()".

NavMesh

El *NavMesh* es el sistema de navegación (*PathFinding*) de Unity. Aquest ens permet donar-li autonomia als nostre *GameObjects* perquè naveguin per l'escena i trobin el camí correcte des d'un punt 'a' fins un punt 'b'. Perquè això funcioni hem d'afegir als *GameObjects* el component *NavMeshAgent*. Aquest els permet utilitzar el *NavMesh* i té molt paràmetres personalitzables: velocitat, acceleració, tamany màxim que pot sobrepassar sense saltar, altura (útil per saber si pot o no passar certes portes) i molt més.

Colliders

És el component encarregat de detectar les col·lisions entre objectes. Són invisibles i poden adoptar moltes formes diferents. Gairebé tots els objectes dels joc necessiten *Colliders* ja que si no fos així serien atravessats per altres *GameObjects* de l'escena. Aquests ofereixen funcions per a accedir a la informació referent als *GameObjects* amb els que interacciona. Aquest component té una variant anomenada *Trigger*, que ens permet detectar col·lisions sense convertir-lo en un objecte sòlid, per tant, permet que altres *GameObjects* el travessin .

Rigidbody

És un component que ens permet rebre i controlar les físiques del *GameObject* al que està adherit. Amb aquest component aconseguirem reaccions més realistes en el moment que rebem alguna força.

Raycast

El *raycast* és, com el propi nom indica, un raig que va des d'un origen a una direcció o punt concret. Quan aquest xoca amb algun *GameObject* ens proporciona informació sobre la col·lisió. El raig és invisible per l'usuari dins el videojoc, tot i que dins de Unity el podem veure.

AudioSource

Aquest component fa la funció d'altaveu dins de Unity, acoblant-se a algun *GameObject* de l'escena. L'*AudioSource* necessita un clip d'àudio i un *AudioMixer* per on reproduir-lo. Cal esmentar que disposa d'un gran número de paràmetres customitzables per ajudar-nos a aconseguir el tipus de so desitjat (Figura 2).

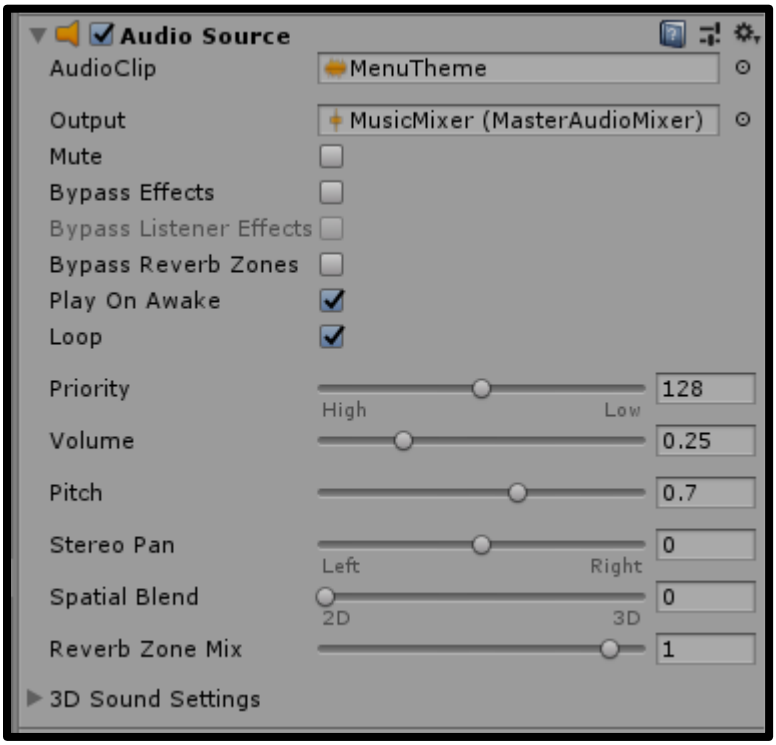

Figura 2: *AudioSource* referent a la música del menú principal.

AudioMixer

L'*AudioMixer*, seguint amb la simetria anterior, fa la funció d'una taula de mescles entre l'altaveu (*AudioSource*) i el clip d'àudio. Aquest ens permet crear nivells de prioritat i/o barreges entre diferents grups. Els grups serveixen per separar els diferents àudios del videojoc i, així, poder aplicar modificacions discriminant per tipus (Figura 3).

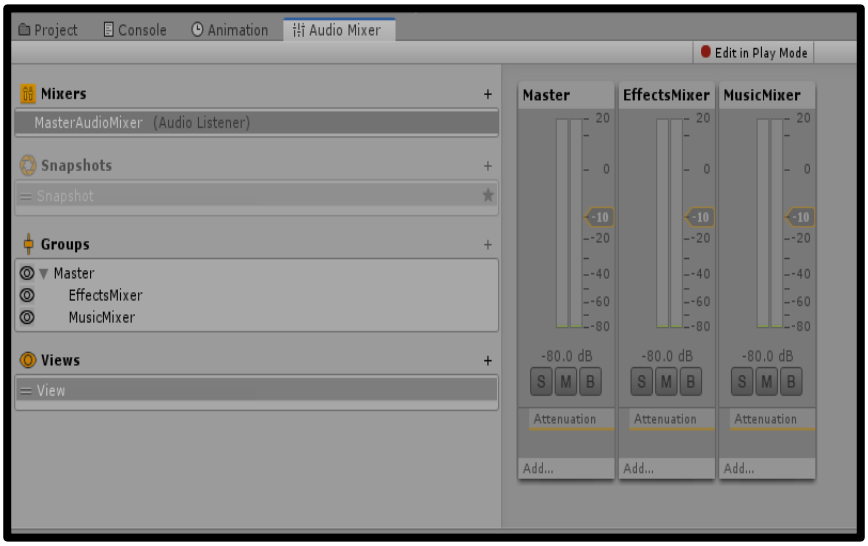

Figura 3: *AudioMixer* amb els 3 grups: *Master*, *EffectsMixer* i *MusicMixer*.

MonoBehaviour

Unity disposa d'una classe anomenada *MonoBehaivour*. Aquesta permet als *scripts* que l'implementen, interaccionar amb l'escena i els *gameObjects* que conté. Per a poder comunicarnos amb l'escena, ens proporcionarà una gran varietat de mètodes per accedir als diferents tipus de components.

Per defecte, aquesta classe disposarà de dues funcions:

Update

Aquesta és una funció que s'executa a cada fotograma. Servirà tant per a rebre inputs del jugador com també per aplicar canvis a l'escena de forma continuada.

Start

El mètode Start s'executara a l'inici de l'escena. S'utilitzarà per a inicialitzar les variables dels *scripts* i/o realitzar tasques quan el *GameObject* es crea.

1.4.2 Blender

Blender és un programa gratuït i molt complet. La seva principal funcionalitat és el modelatge 3D, però també té moltes altres funcions, com les que s'enumeren a continuació: *rigging*, animacions, simulació d'escenaris, pintura digital, cinemàtiques, etc. Nosaltres només l'utilitzarem per modelar, fer el *rigging* i animar (Figura 4).

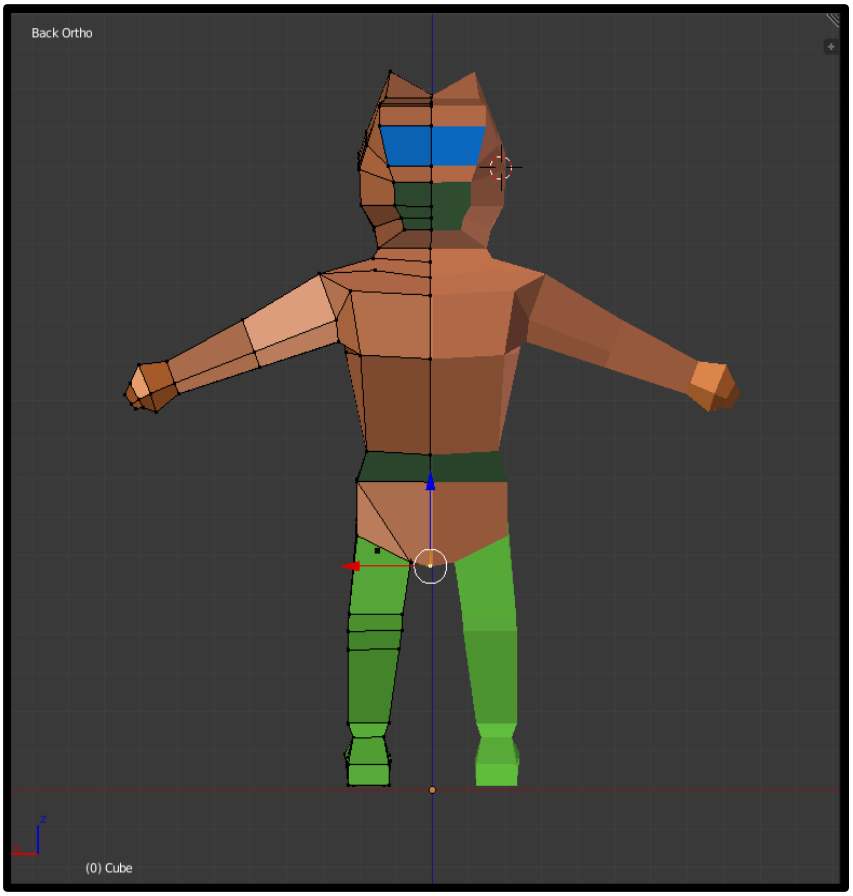

Figura 4: Model 3D del personatge principal.

Modelar:

A l'hora de modelar un objecte es pot fer de diverses maneres. Les següents són tres d'elles:

- Modelar a partir de vèrtex: es creen i connecten vèrtex per crear plans.
- Modelar a partir de plans: es modifica la posició i distància entre vèrtex a partir d'un pla o es talla aquest per a crear-ne de nous i, finalment, connectar-los per a crear figures 3D.

- Modelar a partir de figures 3D: es parteix de qualsevol de les figures 3D que ens proporciona Blender (cubs, piràmides…). Ja amb la figura creada, es podrà deformar i tallar per aconseguir noves figures 3D.

Rigging:

El rigging és la part del modelage 3D que aportarà vida, expressió i moviment als nostres models. En poques paraules, el *rigging* ens permet simular els ossos del personatge utilitzant nodes, així que es fa imprescindible per crear animacions. Aquests nodes disposen d'un pes. Aquest ens indica la quantitat de vèrtex del model, pròxims als nodes, que seran afectats pel seu moviment i/o rotació. Per no perdre l'estructura construida en el moment de fer animacions i aconseguir uns moviments més naturals es poden crear relacions entre els nodes. Aquestes relacions ens permeten definir un node com a fill o pare d'un altre. Aquestes relacions fan que els nodes fill segueixin el moviments del pares i això ajuda a no perdre l'estructura inicial *rig*.

Animar:

Per tal d'animar de forma correcta, com ja s'ha esmentat anteriorment, és necessari un bon sistema de *rigging*. Una animació es composa per un nombre de fotogrames, que defineixen la llargada de l'animació. A cada un d'aquest fotogrames disposarem els nodes de rigging en les posicions desitjades fins a aconseguir una animació fluida. En la Figura 5 podrem veure la llista de nodes a l'esquerra i, a la dreta, la posició assignada a cada fotograma. Com es pot veure en la figura, no és necessari assignar una localització a cada fotograma, ja que Blender ens assigna automàticament les posicions dels nodes entre dos fotogrames on ja s'hagin assignat posicions prèviament.

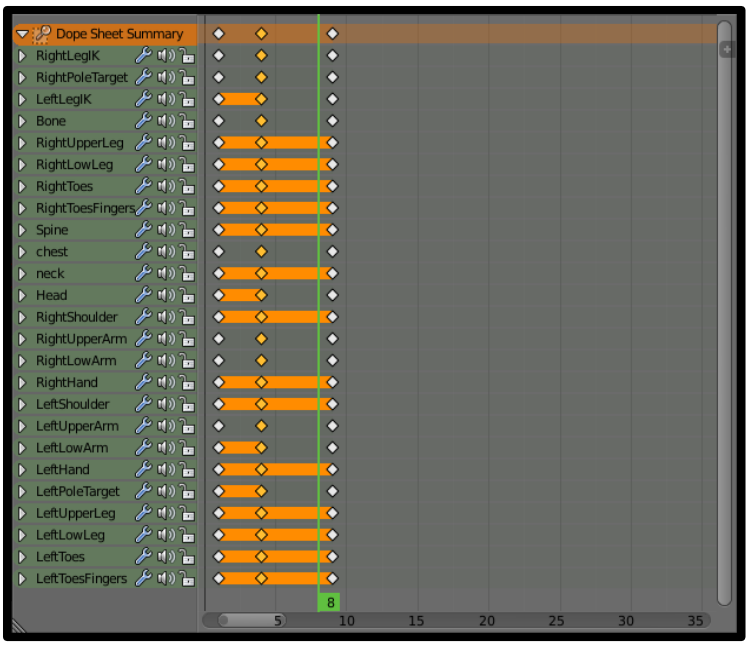

Figura 5: Pestanya de creació d'animació a Blender, on es poden veure tot els nodes a l'esquerra, i a la dreta els fotogrames de l'animació.

1.4.2 Altres programes utilitzats

Audacity

Audacity és una eina molt simple i útil, a més de gratuïta, que serveix per a retocar, tallar o unir arxius d'àudio, canviar l'altaveu per on es reprodueix el so, etc. En aquest projecte s'ha utilitzat únicament per retallar alguns sons.

Paint 3D i Paint Tool SAI

Es tracta de dos programes gratuïts que serveixen per crear o modificar imatges. S'han utilitzat per a petits retocs a imatges, retalls, modificacions del color i canvis en la resolució.

Capítol 2. Implementació

2.1 Managers generals

Per poder controlar de la forma més ordenada possible un joc, és necessari disposar de managers. Aquest son uns *scripts* que controlen tot allò relacionat amb un àmbit concret.

GameManager

Aquest manager és l'encarregat de crear els altres managers, així com d'iniciar una partida, carregar l'escena (mapa) corresponent i inicialitzar tots els objectes (jugador, càmera, menús, etc.) necessari perquè tot funcioni correctament.

AudioManager

Aquest manager és l'encarregat de reproduir i controlar tots els clips de so del joc. Per a fer-ho, disposa d'una referencia a l'*AudioMixer* principal i una llista de "Sounds.cs" (Figura 6).

L'*AudioMixer* del que disposem l'he configurat de tal manera que tenim un grup principal i, dins d'ell, dos grups diferents, *MusicMixer* i *EffectsMixer*. Això ens permet controlar el volum de tot el joc des del mixer principal i, si volem més precisió, podem configurar el volum de la música i el volum dels efectes per separat.

La classe "Sounds.cs" conté informació sobre els àudios: el nom, el clip d'àudio en sí, el volum, si s'ha de reproduir en bucle, un identificador del mixer on s'ha de reproduir i, finalment, una referència al *AudioSource*.

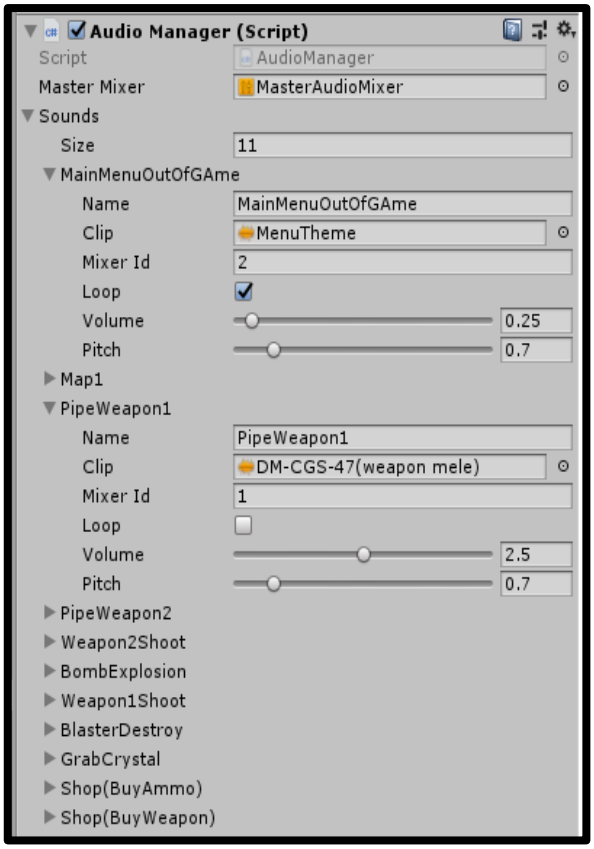

Figura 6: *Script* "AudioManager.cs" on es pot veure la llista de "Sounds.cs".

En obrir el joc i generar-se aquest manager, es crea un *AudioSource* per a cadascun dels sons. A aquest últim li carregarem tota la informació que conté el "Sounds.cs" i el guardarem com a referència en una llista.

Així doncs, quan es vol reproduir qualsevol dels sons es crida al mètode "play(string audioName)" del manager i aquest buscarà dins la llista i ficarà en marxa el *AudioSource* preparat a l'inici. De la mateixa manera que es pot reproduir àudios, també disposa d'un mètode per parar-los anomenat "stop(string audioName)".

SettingsManager

Aquest és el més lleuger dels tres, ja que conté els paràmetres (sensibilitat del ratolí, volum, etc.) modificables des de el menú d'opcions. En el moment en que qualsevol *GameObject* s'inicialitzi si conté algun d'aquest paràmetres, accedirà a aquest Manager per a carregar els valors assignats per l'usuari. Finalment, aquest manager també disposa de la referència a tots els *scripts* que utilitzen paràmetres variables, per a notificar-los si l'usuari els canvia després de que aquests ja s'han inicialitzat.

2.2 Creació del Personatge

El personatge principal és la part més important del videojoc, ja que serà el protagonista en tot moment i, per tant, és la part a la que s'ha dedicat més temps i esforç.

En termes de components, aquest *GameObject* té equipat un *characterController* que té un *collider* en forma de cilindre que envoltarà el model 3D del personatge. Aquest serveix perquè el personatge col·lisioni amb la resta d'objectes, i també tindrà equipat un *animator* per controlar les animacions, un *Rigidbody* per a poder aplicar-li forces en qualsevol moment i, finalment, els *scripts* pertinents.

El personatge a nivell de programació consta de quatre parts principals: el moviment, les armes, la vida i la càmera. Sabent això, s'ha dividit el personatge en quatre grans *scripts*: "PlayerStatsManger.cs" (*2.2.1 Manager d'estats*), "Movement Manager.cs" (*2.2.2 Manager de moviment*), "WeaponManager.cs" (*2.2.3 Manager d'armes*) i "PlayerCamera.cs" (*2.2.4 Càmera*). Aquests tres primers seran els *Managers*, els encarregats de controlar tot el referent als seus àmbits. Ens permetran comunicar tots els *scripts* d'una forma ordenada i escalable.

2.2.1 Manager d'estats

Aquesta classe serà l'encarregada de controlar i guardar tota la informació sobre els estats del personatge, així com altra informació rellevant com podrien ser els cristalls acumulats, l'experiència actual i el nivell en el que es troba. Per tant, si qualsevol *script* necessita informació d'aquest tipus, la demanarà a aquest manager(Com es pot veure a UML de la Figura 7).

Aquest manager es troba a la classe "PlayerStatManager.cs", que estén d'una classe més general anomenada "StatManager". Aquesta tindrà implementades totes les funcionalitats imprescindibles per a donar-li vida a qualsevol personatge. Aquestes funcionalitats bàsiques es podrien resumir en dues: gestionar la vida i l'estat en què es troba l'objecte portador del *script*. Per tant, el simple fet d'afegir aquest *script* a un objecte ja el farà interactiu per a qualsevol acció dins el joc que apliqui estats o modificacions en la vida.

La classe "PlayerStatManager", a part de l'esmentat anteriorment, també afegirà gestió d'experiència, nivells i monedes. També substitueix algunes funcions de la classe pare com "hurt()", on implementarà una animació de rebre mal, i "die()", que també substitueix l'acció per defecte (destrueix l'objecte en el moment de morir) per una animació de mort.

Estats

Els estats són, com el propi nom indica, estats alterats del personatge, com podria ser la immobilitat, la inhabilitació de les armes, una empenta, l'enverinament, l'alentiment, etc. Seran molt útils per atorgar varietat d'enemics, ja que, es podrà potenciar de maneres diferents les seves armes i diferenciar-los de la resta.

Per a poder controlar la interacció entre tot els possibles estats alterats la classe "StatManager" disposa d'una llista dels estats rebuts. D'aquesta manera com que tots els estats s'aplicaran des del mateix *script*, es podran gestionar els efectes d'una forma més ordenada que evitarà la creació de bugs. Qualsevol de la resta de *scripts* de l'escena podran afegir-li estats sense preocupar-se de les condicions de l'objectiu.

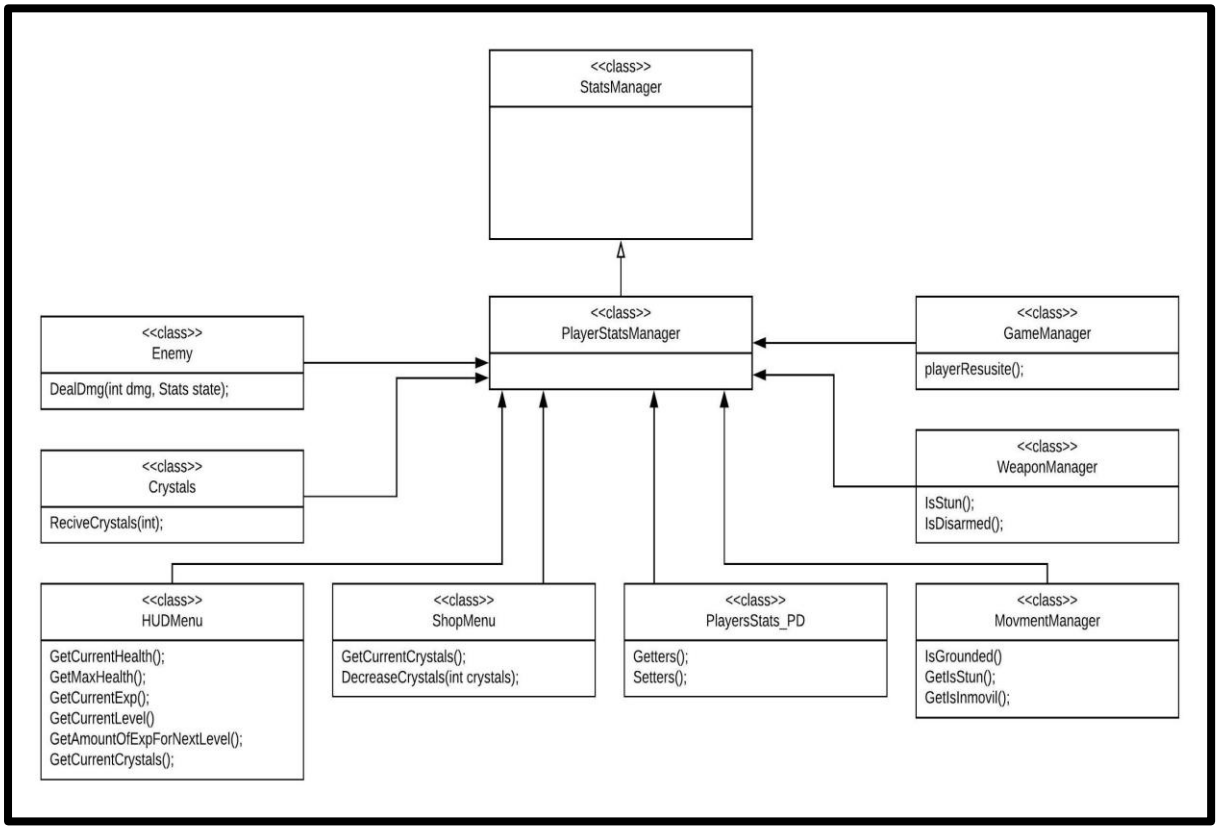

Figura 7: UML que mostra tots els *scripts* que interaccionen amb el manager d'estats.

2.2.2 Manager de moviment

Es troba a la classe "MovementManager.cs", que conté tot el necessari per controlar el moviment del personatge i les físiques que s'aplicaran sobre ell.

Per facilitar la creació del moviment del personatge s'ha utilitzat un component de Unity que es diu *characterController*. Aquest ens ofereix moltes funcionalitats per a facilitar la implementació de les funcionalitats del personatge.

Moviment

Per a fer que el personatge es mogui s'ha utilitzat un mètode anomenat "Move()", que és part del *characterController*. Per defecte, el personatge sempre rotarà cap a la direcció on volem caminar. Tot i que això és molt útil per moure's i posicionar-se al lloc desitjat, en el moment de lluita contra enemics no és del tot útil, ja que es difícil atacar i caminar endarrere o cap als laterals per esquivar projectils. Com a solució, s'ha implementat una funcionalitat que permet bloquejar la rotació del personatge. D'aquesta manera, el personatge sempre estarà mirant en la direcció de la càmera i ens permetrà disparar endavant mentre ens movem en la direcció desitjada.

Gravetat

Hem contingut el funcionament de la gravetat en un mètode anomenat "gravity()" i una variable global anomenada "velocity_y". Aquest metode sumara a cada fotograma velocitat negativa i l'aplicarà al personatge. D'aquesta manera si no estem tocant al terra caurem cap a baix. Com que no volem que s'acomuli velocitat negativa infinitament hem afegit tant un límit de velocitat negativa com un reset d'aquesta quan el personatge estigui al terra.

Per ajudar al manager a controlar si el personatge està en contacte amb el terra s'ha creat la funció "IsGrounded".

Tot i que el *characterController* de Unity ens ofereix moltes facilitats, alguns dels mètodes de que disposa no funcionen del tot bé. La funció "isGrounded" concretament, és una d'elles. Aquesta funció és la que dóna informació sobre si el personatge està tocant al terra o no. Aquesta funció és molt important per saber quan el personatge pot saltar i quan s'ha d'aplicar gravetat sobre aquest.

Per a detectar el terra es van estudiar diverses estratègies i, finalment, vaig decidir implementar el meu propi "isGrounded()". Vaig crear dos cercle a la base del personatge composat per vint raycasts, repartits en dos cercles i dirigits dels peus de personatge cap al terra (Figura 8). Quan en algun d'aquest raycasts es detecta una col·lisió a una distància més petita que la preestablerta es dedueix que el personatge està tocant el terra.

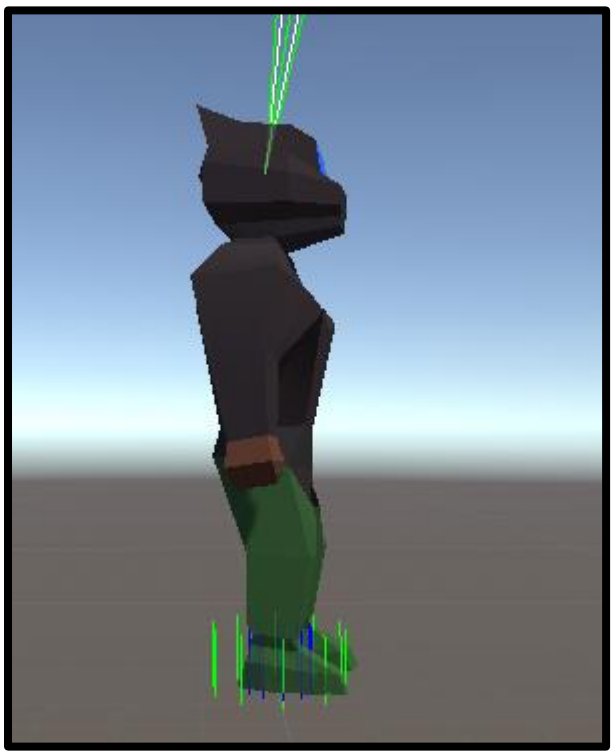

Figura 8: *Raycast* de la funció “isGrounded()”.

Salt

En un joc de plataformes és imprescindible el salt del personatge. En el nostre cas, el personatge podrà fer un total de dos salts seguits sense necessitat de tocar al terra. Per a controlar el nombre de salts tenim una variable que conté el número de salts màxims possibles. Cada vegada que el personatge salta, es restarà un salt d’aquesta variable fins a un màxim de zero, on no es podrà saltar més. Automàticament, es tornarà a assignar el número màxim de salts quan el personatge toqui el terra. Com ja s’ha esmentat en l’apartat anterior, el salt va molt relacionat amb la gravetat, doncs el funcionament és tan simple com assignar un número positiu a la “velocitat_y” (la variable que també modifica la funció “gravity()”). La funció “gravity()” ja s'encarrega d’aplicar la nova velocitat en l’eix vertical i propulsar al personatge.

2.2.3 Manager d'armes

Per a poder gestionar un gran nombre d'armes de forma fàcil, ordenada i escalable s'ha creat un *script* anomenat "WeaponManager.cs"(UML dels scrips que interactuen amb ell Figura 9). Aquest *script* s'encarregarà de gestionar tota la informació sobre les armes, d'activar-les i desactivar-les, així com d'utilitzar-les. Per a fer-ho, el *script* disposarà d'una llista on es guardaran totes les armes, un identificador de l'arma equipada actualment i una llista dels possibles objectius.

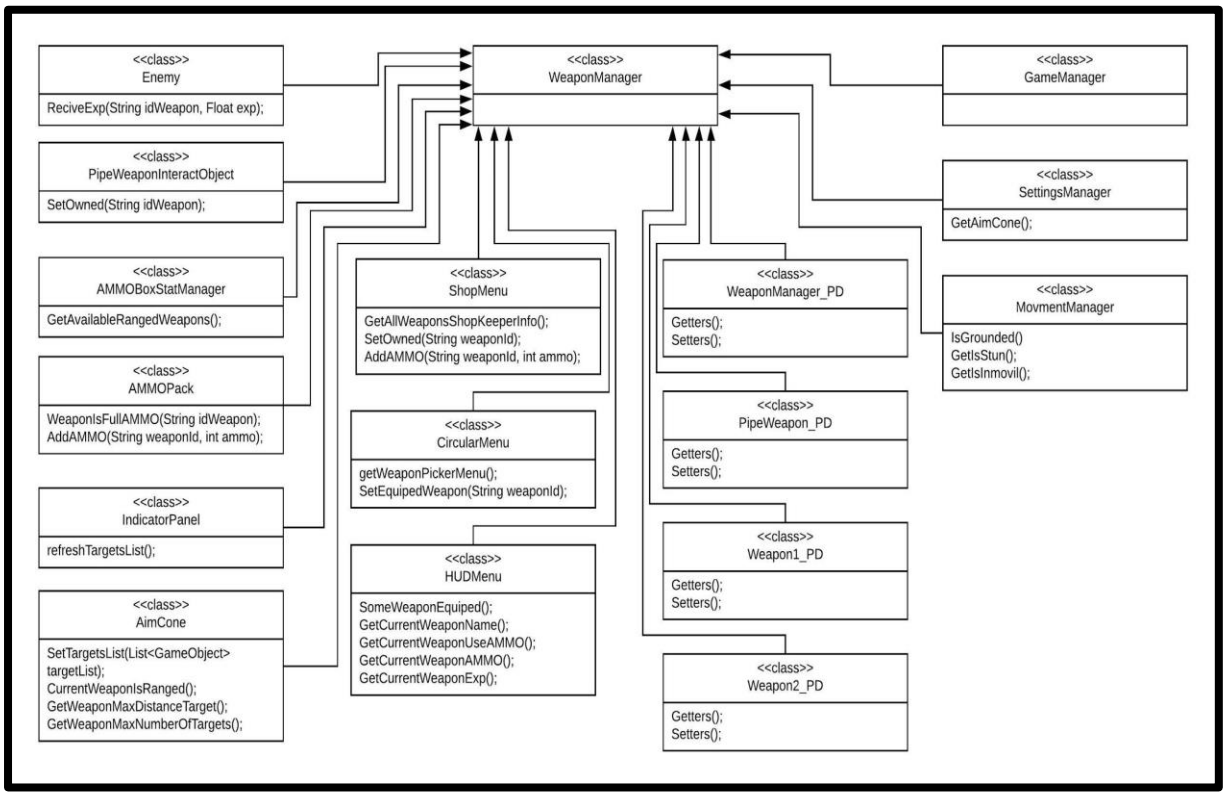

Figura 9: UML que mostra tots els *scripts* que interaccionen amb el manager d'armes.

Auto apuntat

Per a un joc que utilitza una càmera en tercera persona no es fàcil apuntar a un enemic de forma precisa. Per tant, per a facilitar aquesta tasca, s'ha implementat un sistema d'ajuda a l'apuntat.

Per a aconseguir-ho, he creat un figura en forma de con, que en tot moment es trobarà davant del personatge de forma invisible (Figura 10). Aquesta figura utilitza un *Trigger* que farà que no col·lisioni amb la resta d'objectes, però sí que detectarà si algun dels enemics entra dins la seva àrea. Aquest con respon al moviment del ratolí, però només es pot moure de forma vertical. D'aquesta manera, podrà seguir el moviment de la càmera i mantenir-se al davant del personatge, de manera que només trobarà objectius que estiguin davant del personatge.

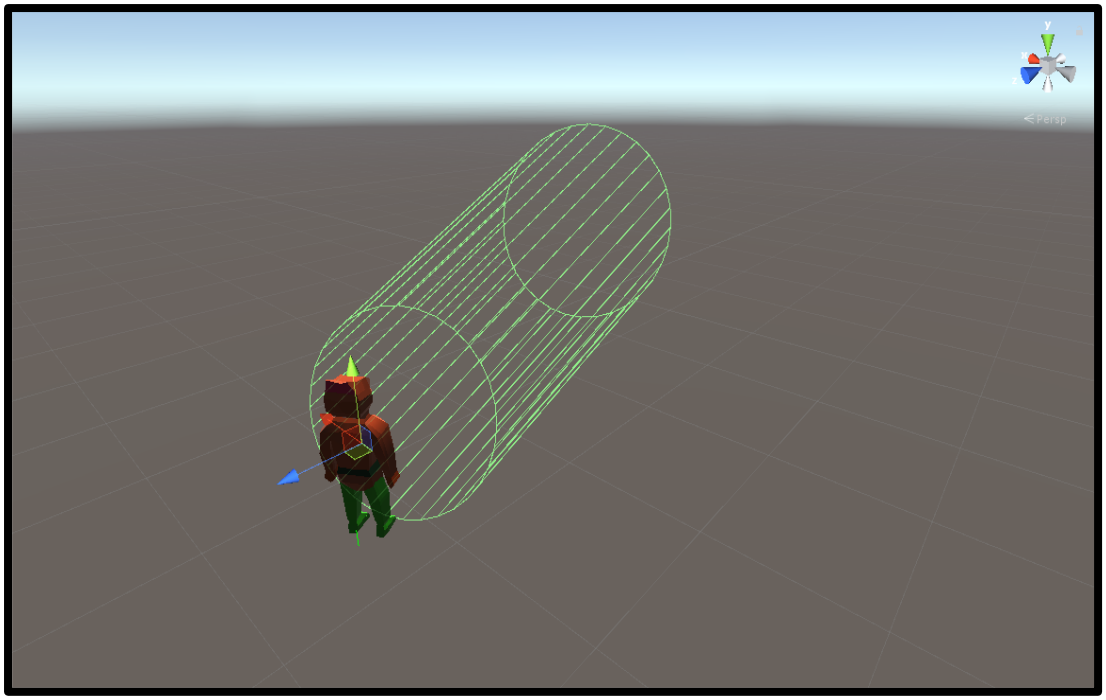

Figura 10: Con encarregat de detectar enemic del sistema d'auto apuntat.

Aquest con disposarà del *script* "aimCone.cs". Aquest *script* serà l'encarregat de generar una llista d'enemics vàlids. A continuació, s'explica com s'obté aquesta llista.

Per poder obtenir la llista es crearà un *HashSet* de *GameObjects*. S'utilitza un *HashSet* perquè permet afegir i eliminar enemics en qualsevol moment de forma paral·lela.

Cada cop que algun enemic entri o surti de l'àrea (o mori) s'actualitzarà el *HashSet*. Quan el "WeaponManager" demana la llista es comprovaran tres coses: que no retornem una llista d'enemics amb més enemics dels que l'arma equipada pot auto apuntar, que aquests enemics estiguin dins una distancia màxima i, finalment, que els enemics no estiguin amagats darrere d'un obstacle (Podem veure millor el funcionament al diagrama de la Figura 12). Per comprovar aquesta última, utilitzarem un raig *raycast* des del jugador a l'enemic. Si el primer objecte en col·lisionar és l'enemic, sabem que tenim camp visual lliure entre els dos i, per tant, serà un objectiu vàlid (Figura 11).

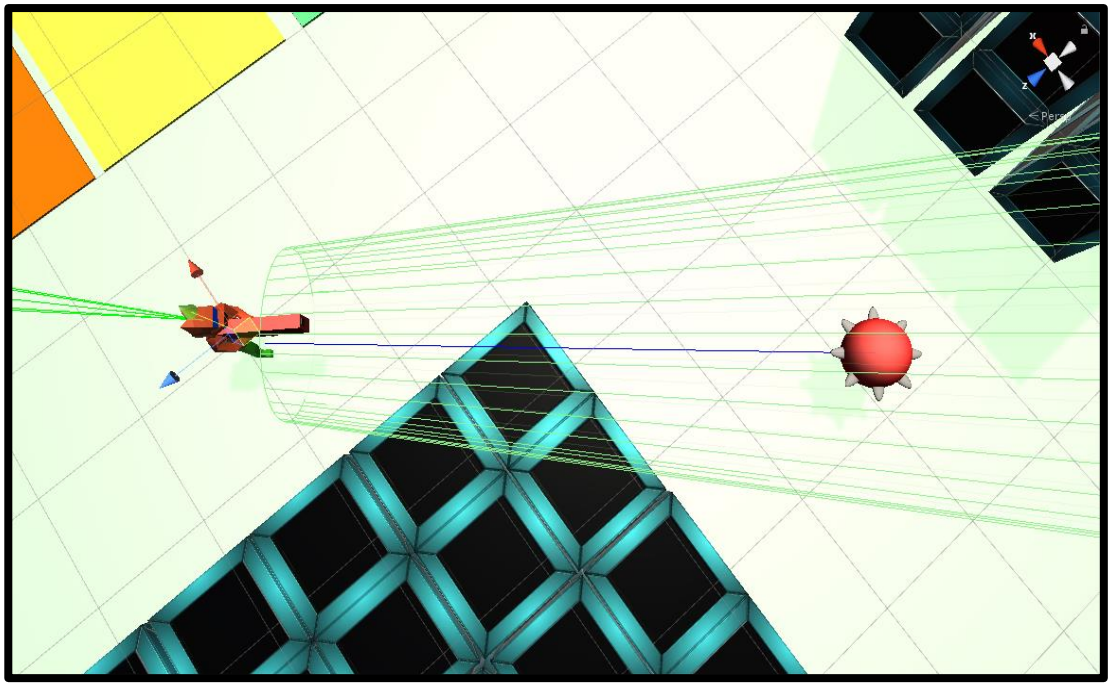

Figura 11: *Raycast* del sistema d'auto apuntat per reconeixement d'obstacles.

Finalment, s'ordenaran per proximitat i s'enviarà la llista d'objectius al "WeaponManager.cs". D'aquesta manera, sabrem que el primer de la llista tindrà més prioritat que la resta. Per a donar *feedback* a l'usuari dels objectius detectats, utilitzarem el panel HUD (*Apartat 2.4.7 HUD*).

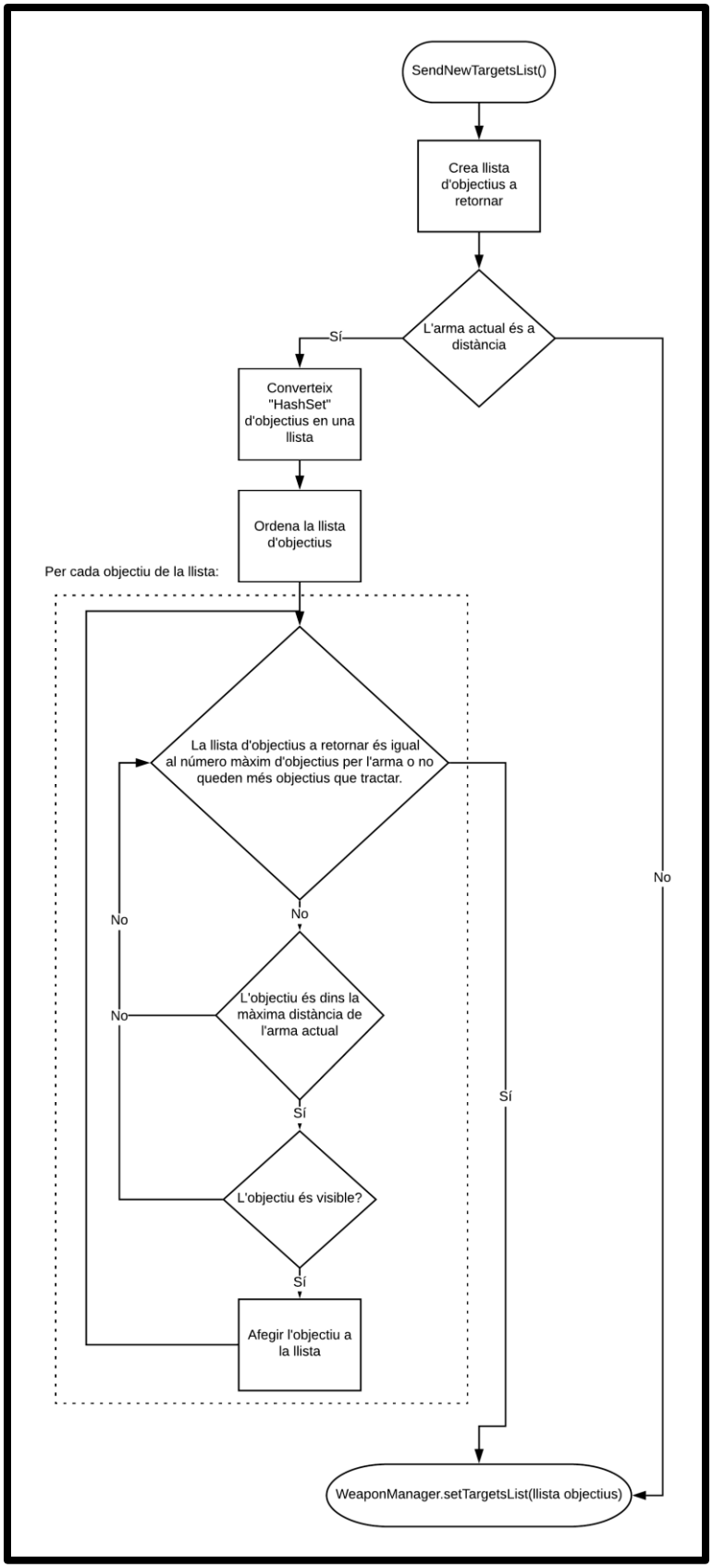

Figura 12: Diagrama de flux del sistema per generar la llista d'objectius.

2.2.4 Càmera

La càmera és una part molt important d'un videojoc, ja que és la finestra a través de la qual el jugador interacciona amb tot el contingut del joc. Aquesta és una càmera en tercera persona i es controla amb el ratolí.

El *script* que controlarà la càmera es diu "PlayerCamera.cs". Aquest té dues funcionalitat principals:

- Moviment de la càmera

Aquesta és la més senzilla de les dos, ja que Unity té facilitats per detectar la posició del cursor dins la pantalla. Amb aquests inputs mourem la càmera, sempre sense perdre el focus del personatge.

- Obstrucció de visió

Un dels principals problemes de les càmeres en tercera persona sorgeix quan tenim algun obstacle que s'interposa entre la càmera i el nostre personatge, i no ens permet veure el que estem fent. Per solucionar-ho, he dissenyat un sistema basat en l'ús de raigs *raycast* en el que cinc raigs disposats entre el jugador i la càmera (Figura 13) detectaran la col·lisió d'obstacles i faran que aquesta es mogui fins al punt on s'ha detectat la col·lisió, de manera que tindrem una visió neta i l'obstacle no ens molestarà. Quan els raigs ja no detectin col·lisions, la càmera tornarà automàticament i de forma gradual a la distancia original. Que la càmera no s'apropi de forma instantània i torni de forma progressiva evita moviments bruscos de la mateixa quan troba un obstacle amb formes irregulars.

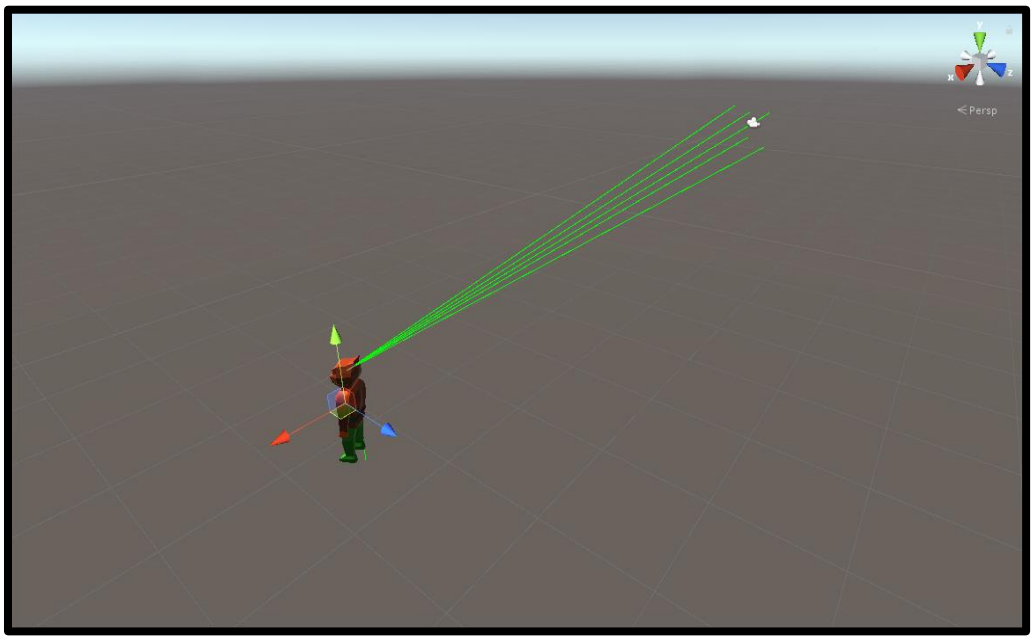

Figura 13: *Raycast* entre la càmera i el personatge.

2.3 Armes

Les armes seran diferents a altres videojocs *shooter*, ja que, en aquest joc totes les armes tindran experiència i nivell propi, i algunes també habilitats especials. Quan l'arma evolucioni es milloraran el atributs i/o s'afegiran noves característiques.

Els *scripts* de totes les armes estenen d'una classe anomenada "Weapons.cs", que implementa la classe abstracta "IWeapons.cs." Aquesta contindrà tots el mètodes necessaris per a comunicar-se amb el manager. Amb aquest disseny podem comunicar-nos des del "WeaponManager.cs" a totes les armes amb els mateixos mètodes. Amb aquesta abstracció podrem fer modificacions i crear tantes armes com es vulgui sense necessitat de modificar cap línia de codi de la resta de *scripts*.

Totes les armes, en el moment de ser utilitzades, si les seves característiques ho permeten faran ús de la llista d'objectius esmentada en l'apartat *2.1.3 Manager d'armes*.

2.3.1 Arma cos a cos

Aquesta és una arma especial, ja que ha d'estar disponible en tot moment. Per aquest motiu, té assignat un botó solament per ella. Aquesta arma té una combinació de tres atacs, que s'efectuarà al prémer tres vegades seguides la tecla d'atac. Els atacs d'aquesta arma aplicaran un estat "push" (empenta). Aquest estat ens permetrà pegar de forma més segura a curta distància.

Ja que és una arma especial, té una interacció especial quan l'utilitzem dins el segon salt. Propulsarà amb força al personatge cap al terra destruint i empenyent els objectes i enemics dins la trajectòria (Figura 14).

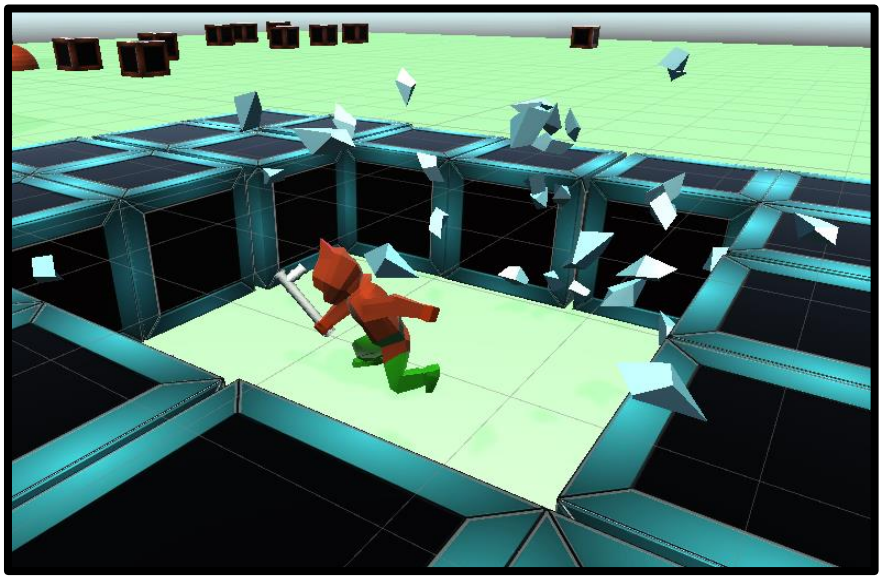

Figura 14: Atac aeri amb l'arma cos a cos.

2.3.2 Pistola làser

Aquesta pistola dispara làsers amb una cadència bastant alta, però de poca potència. La implementació és pràcticament la que utilitzen les armes convencionals, simplement donarà una direcció i una velocitat al projectil.

2.3.3 Pistola llença-bombes

Aquesta segona arma ja és més peculiar. De cadència baixa i gran potència en àrea d'efecte, llença bombes en una trajectòria parabòlica que, en col·lisionar, crearan un explosió que aplicarà l'estat "KnockUp" als enemics, fet que causarà que els empenyi en la direcció contrària. En el moment de disparar l'arma, crea una bala i la configura, enviant-li la posició de l'objectiu, l'angle de sortida i la velocitat, de manera que aquesta traçarà la paràbola de forma correcta.

2.3.4 Projectils

Tots el projectils tindran un identificador d'arma, un mal i/o un estat assignat per l'arma que el dispara. En el moment que el projectil col·lisiona amb els objectes de l'escena, el primer que farà és comprovar si aquest disposa d'algun *script* derivat de "StatManager.cs" i, si és així, aplicarà el mal, l'estat del projectil i, finalment, es destruirà. En cas que col·lisioni amb un enemic, aquest enviarà experiència a l'arma que ha disparat el projectil (a través del "WeaponManager").

Per a poder assegurar un bon rendiment, els projectils tenen temps de caducitat. Un temps de caducitat ens assegura que, si en algun cas el projectil no col·lisiona amb cap objecte, el projectil es destrueix al cap d'uns segons i, així, s'evita que projectils perdut consumeixin recursos inceseraris.

Per als projectils, utilitzarem una interfície anomenada "Proyectile_interface" que permet assegurar que els altres *script* puguin interactuar amb ells sense preocupar-se del tipus de projectil del que es tracta.

Projectil Blaster

Es tracta d'una bala senzilla, ja que utilitzarà un *script* molt senzill, que l'únic que fa és esperar a col·lisionar amb un objecte.

Projectil Bomba

Aquest projectil, a diferència de l'anterior, tindrà un recorregut en forma de paràbola. Per aconseguir aquest recorregut, l'arma que el dispara li enviarà les tres dades següents: la

posició de l'objectiu, l'angle i la velocitat. Amb aquesta informació, podrà calcular la força de gravetat que cal aplicar al projectil perquè arribi a l'objectiu amb la velocitat adient. En el moment en que col·lisoni amb qualsevol objecte, crearà una ona expansiva que farà mal i empenyerà als enemics situats en un cert radi al voltant seu, i finalment es destruirà.

2.4 UI

Un sistema intuïtiu i fàcil és imprescindible per a que el jugador se senti còmode i pugui moure's per la interfície de forma ràpida sortint el menys possible de l'ambientació que crea el videojoc. Per a poder construir qualsevol component de UI és necessari fer-ho sobre un "canvas" (llenç) . Així mateix, per a poder veure i interaccionar amb ell, és necessària una càmera.

2.4.1 Menú inicial / Menú dins del joc

Tant el menú principal com el menú de dins el joc disposen dels seus respectius managers, "MainMenu.cs" i "MenuManager.cs" respectivament. Ja que la seva principal funcionalitat és oferir accés al jugador a la resta de menús disponibles, no disposen de grans funcionalitats, més enllà de controlar les interaccions entre els altres menús. Tot i això, el menú principal sí que disposa d'una funcionalitat adicional: la d'iniciar una nova partida (Figura 15).

Figura 15: Menú principal.

2.4.2 Menú Guardar i Carregar

Aquests dos menús seran els encarregats de guardar i carregar tota la informació referent a la nostra partida. El joc disposa de quatre ranures on es podrà guardar i carregar el nostre progrés (Figura 16). Per a guarda la informació d'una manera segura i impedir així als *hackers* modificar el joc, s'ha utilitzat un *BinaryFormatter*. Aquest és l'encarregat de convertir a binari la informació que volem guardar, de manera que modifica els fitxers que genera el joc perquè aquests no es puguin modificar tan fàcilment. Aquest formatejador és fàcil de fer servir i eficaç, però, en contrapartida, només pot convertir variables de tipus primitius (ints, strings, floats, etc.). Per tant, s'hauran de transformar algunes de les variables per a poder guardar-les.

Per a cada una de la informació que s'ha de guardar (dels diferents *scripts*), es crearà un segon *script*, el qual serà l'encarregat de fer la conversió de tipus (de qualsevol dels formats que utilitza Unity i C# a tipus primitiu). Així doncs, aquest *scripts* implementaran la interfície "IPersistenceData", que els forçarà a implementar els mètodes "load()" i "save()". Aquests seran els que es faran servir des de la classe "PersistenceManager.cs".

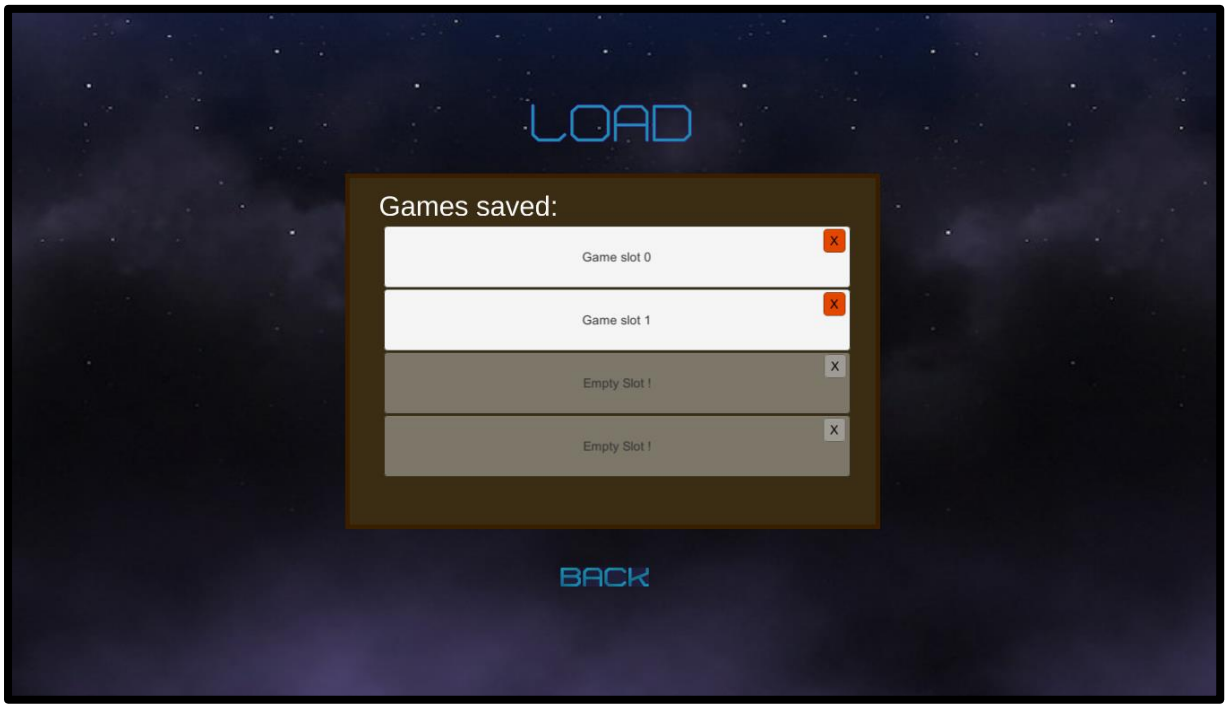

Figura 16: Menú de càrrega de partides.

La classe "PersistenceManager.cs" serà l'encarregada de modificar, carregar, guardar i borrar les partides. Totes les partides tindan un directori propi amb múltiples fitxers, un per a cada tipus d'informació. En aquest punt, em vaig adonar que part de la informació que s'ha de guardar es podia destruir abans que el jugador guardés la partida i, per tant, això suposaria la pèrdua d'aquesta.

Per exemple, si ens fixem en la informació que conté un mapa (portes ja activades, punts de control desbloquejats, etc.) aquesta es perdrà en el moment en el que canviem d'escena. Ja que cada mapa està contingut en una escena separada, la informació d'aquesta serà destruïda quan ens moguem d'un a un altre, el que significa que s'ha d'haver guardat prèviament.

Ja que no es vol obligar al jugador a guardar abans d'abandonar cada mapa, per a solucionar-ho, en el moment de carregar o començar una partida, es genera un directori auxiliar on s'acumularà tota la informació que s'ha de conservar. Tots els *scripts* que podrien ser destruïts sense guardar el progrés, creen en aquest directori el seu fitxer d'informació abans de destruir-se. Quan el jugador decideixi guardar la partida, s'accedirà a aquest directori auxiliar i es mourà tot el seu contingut a la ranura escollida.

2.4.3 Menú d'opcions

Aquest disposa de la interfície per representar tot els paràmetres dels *scripts* que es poden personalitzar i serà controlada pel *script* "SettingsManager.cs". Aquest panell disposa d'un *ScrollView*, on apareixen tots els paràmetres, i tres botons: el primer retorna tots els paràmetres als valors originals, el segon és per a sortir i descartar els canvis, i el tercer ens permet sortir i preservar el canvis (Figura 17). Quan es prem aquest darrer botó, acciona un mètode del *script* "SettingsManager" (Apartat *2.1 Managers generals*) que notifica la resta de *scripts* que ho requereixin per a que actualitzin els parametres.

Figura 17: Menú d'opcions.

2.4.4 Menú de confirmació

El menú de confirmació és, simplement, un panell amb una pregunta i dos botons de resposta (Figura 18). Aquest està controlat pel *script* "ResponceDialogMenu.cs", on s'implementa el mètode "openBinaryQuestion()", que és accessible des de qualsevol altre *script*. Aquest mètode rebrà dos valors per paràmetre: un string, el text de la pregunta que es vol fer a l'usuari, i un segon paràmetre que serà una referència a una funció. Aquesta funció ha de tenir un format específic: que accepti un paràmetre booleà d'entrada. Quan el jugador seleccioni una de les respostes, s'utilitzarà aquesta funció per enviar el feedback directament al *script* que ha formulat la pregunta.

Figura 18: Pantalla de confirmació.

2.4.5 Menú de la botiga

El menú de la botiga s'obrirà quan el personatge interaccioni amb el botiguer. El menú de la botiga s'obrirà i apareixeran dues seccions en pantalla: "ScrollView" i "ItemView". El "ScrollView" contindrà, un botó per a comprar tota la munició disponible i, tot seguit, un botó per a cada una de les armes disponibles a la botiga. En fer clic en un dels botons de les armes, s'obrirà dins la segona secció un panell amb informació de l'arma seleccionada. Un cop l'arma seleccionada és ja propietat del jugador, es mostrarà un input de text (inicialment amb el número màxim de munició que pot comprar l'usuari amb els cristalls de que disposa) on l'usuari podrà seleccionar el número de munició que vol comprar i on es mostrarà quants cristalls costaria comprar-la. D'altra banda, si l'arma no està en propietat, es mostrarà el preu de l'arma i una breu descripció. Així mateix, tant si s'ha de comprar l'arma seleccionada com la seva munició, apareixerà un botó a la part inferior de la secció per a realitzar la compra (Figura 19).

Aquest menú està controlat pel *script* “Shop.cs”, que és l’encarregat de generar tots els botons necessaris quan el menú de la botiga s'obre. Primerament, es calcula el preu de tota la munició disponible i, a posteriori, es crea el primer botó (el que ens permet comprar tota la munició d’un sol clic). En segon lloc, es crearan els botons de cada arma. Cada botó contindrà la informació de la respectiva arma i també una referencia a la funció que ha de cridar el mètode “click()” del botó. Aquesta funció obrirà la zona de “ItemView” amb una de les dues versions que s’ha explicat anteriorment.

En qualsevol dels dos casos, tant si es compra l’arma, com si es compra munició, es resten els cristalls gastats al manager “PlayerStatManager.cs” i, d’altra banda, es notifica al “WeaponManager.cs” perquè modifiqui la informació de l’arma en qüestió (sumar munició o canviar l’estat de l’arma a “en propietat”).

Figura 19: Menú de la botiga.

2.4.6 Sistema de Diàlegs

Aquest menú consta de dos panells, un que mostra el nom del narrador i un segon que mostra els missatges (Part inferior de la Figura 20). Controlat pel *script* “DialogManager.cs”, aquest rebrà un objecte “DialogInfo.cs” quan es vulgui iniciar un diàleg. Aquest objecte conté la següent informació: nom de l’autor, id del autor, tipus de conversa i el text que es vol reproduir. El id ens permetrà identificar a l’autor i, el tipus de conversa, escollir entre les diferents opcions de representar el text. Actualment, n’existeixen dues: una que pausa la partida quan es reprodueix el diàleg i una que reprodueix les frases del narrador sense bloquejar la partida i les destrueix passat uns segons de forma automàtica.

Per a cada frase rebuda s'inicia un procés per escriure-la lletra a lletra i, així, es crea un diàleg molt més dinàmic. Si volem carregar-la de cop, n'hi haurà prou amb prémer qualsevol tecla (excepte "Esc"), que cancelarà la funció i l'imprimirà de cop. Per passar al següent bloc de text, és necessari que l'actual estigui completament escrit. Quan ja estigui complet, prement qualsevols tecla (excepte el "Esc") passarem al següent. Per defecte, si intentem passar de bloc però no n'hi ha més, es tancarà el menú.

Figura 20: Imatge de l'interior del videojoc on es pot veure el menú de diàlegs i el HUD.

2.4.7 HUD (Head-Up Display)

Aquest menú té com a finalitat representat un seguit de dades bàsiques del personatge (Part superior de la Figura 20). Aquestes es mostraran en pantalla en tot moment. El *script* que el controla es diu "HUDMenu.cs", i és un dels menús més senzills, ja que només ha de demanar la informació als respectius managers i mostrar-la.

Panel d'auto apuntat

Dins del HUD es troba un altre *script* molt important: el "IndicatorPanel.cs", que és l'encarregat de generar, col·locar i reutilitzar, sempre i quan sigui possible, els símbol d'auto apuntat sobre els objectius de la pantalla.

2.4.8 Selector d’armes

El selector d’armes s’utilitzarà contínuament, ja que la seva funció és seleccionar l’arma que volem equipar. Aquest menú està composat per un total de vuit botons, organitzats en forma de “dònut” (com un gràfic de pastís), on cadascun representarà una arma (Figura 21). Per a seleccionar-ne una, obrirem el menú, mourem el ratolí fins a sobre de l’arma que volem seleccionar i, sense necessitat de clicar, tancarem el menú. El *script* encarregat de que això funcioni és el “CircularMenu.cs”, que en obrir-se, pintarà tots els botons segons allò que vulgui indicar: si aquell botó té una arma assignada, el pintarà de color turquesa i, en canvi, si no és així, el pintarà de color gris. Per poder seleccionar l’arma, sempre que aquest *script* estigui actiu, controla la posició del ratolí. Amb la posició X i Y del ratolí calcula l’angle respecte el centre del menú i així sabem quin dels botons s’ha seleccionat. Per a fer-ho més visual, el botó seleccionat canviarà el color en passar el ratolí per sobre.

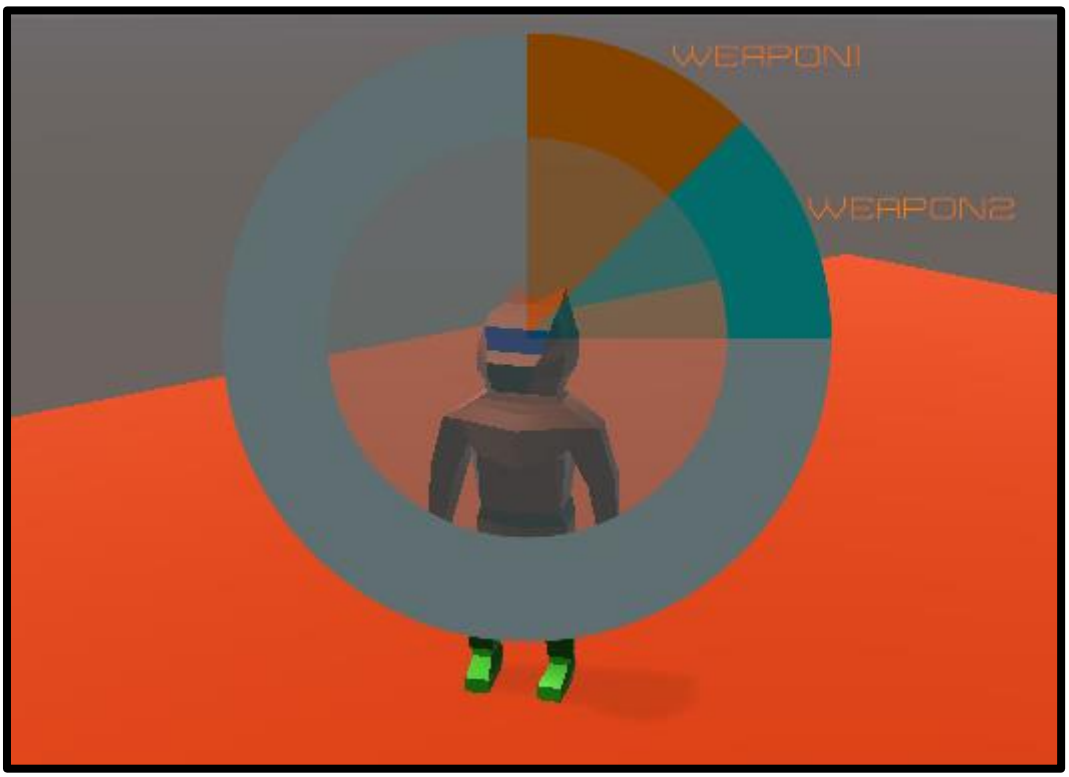

Figura 21: Menú circular de selecció rapida d’armes.

2.5 Enemics

Tots i cadascun dels enemics consten de dos *scripts*. Un primer ens servirà per a controlar els estats i la vida. Aquest extendrà de la classe pare "StatManager.cs". En un segon, s'hi implementarà tot el comportament de l'enemic: tant el moviment sobre el mapa, utilitzant el *NavMeshAgent* i el *NavMesh*, com l'estratègia amb la que atacarà al jugador.

Tots els enemics implementen una estructura similar de components. De la mateixa manera que el personatge principal, aquests estan equipats amb un *Collider* per a detectar les col·lisions i ocupar un espai físic dins el mapa, un *Rigidbody* per aplicar-li forces, un *NavMeshAgent* personalitzat per cadascun dels enemics i finalment els respectius *scripts*.

2.5.1 Enemic cos a cos

Aquest enemic és el més senzill de tots, ja que aplica dany quan col·lisiona amb el jugador (Figura 23). Començar a perseguir-lo en el moment en que aquest entra dins el seu camp de visió (una distancia configurable). Quan aquest aconsegueix atrapar-lo, li aplica un impacte que li restarà vida i l'empenyerà enrere.

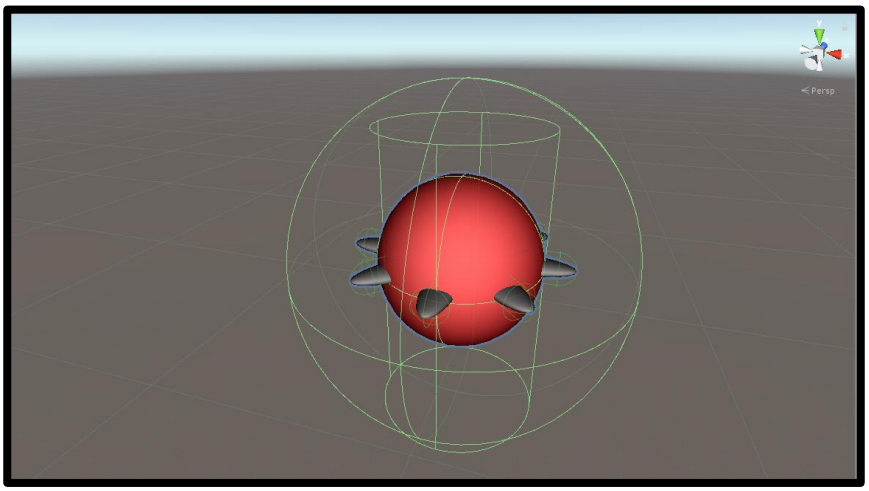

Figura 22: Model de l'enemic cos a cos.

2.5.2 Enemic tanc

Aquest enemic no és res més que una variant de l'enemic cos a cos. Tot i que tècnicament parlant té exactament el mateix funcionament, disposa de tres modificacions addicionals: un augment de tamany, un augment de la seva vida i un efecte complementari, que consisteix en un estat d'immobilitzar a l'atacant durant un breu període de temps.

2.5.3 Enemic a distància

Aquest enemic és més complex que l'anterior. Igual que l'enemic cos a cos, aquest també perseguirà al protagonista quan entri dins el seu camp de visió, però no arribarà a col·lisionar amb ell. Quan arribi a la distància on es troba el seu rang de tir, menys un offset, pararà de moure's i, llavors, començarà a disparar (Figura 23). La funció d'aquest offset és donar a l'enemic un marge per a disparar, abans que el personatge surti de l'àrea. Sense aquest offset, en la majoria dels casos, ja que el personatge està en continu moviment, li seria impossible mantenir-lo dins el rang d'atac i mai tindria oportunitat de disparar.

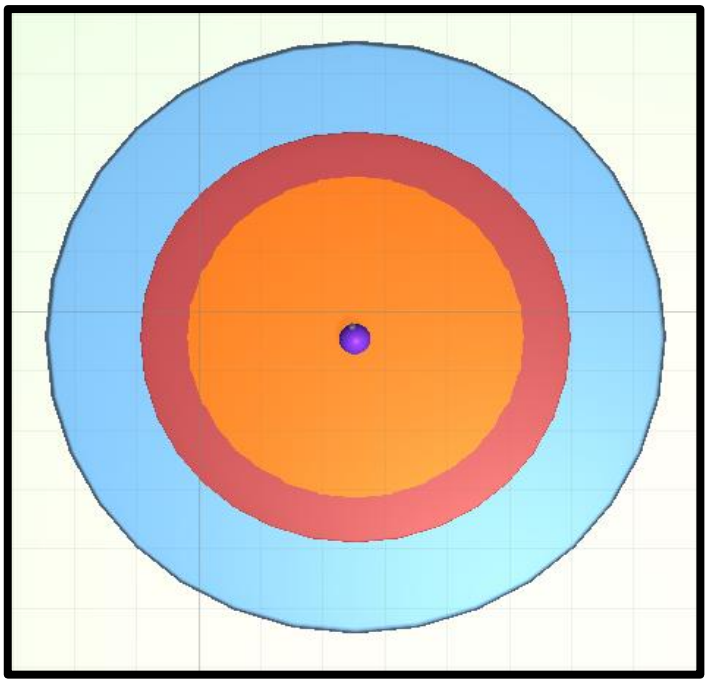

Figura 23: Diferents zones d'acció del enemic a distància (Zona blava: àrea d'atac; zona vermella: àrea de tir; zona taronja: àrea de disparar, menys l'offset).

2.6 Mapes

Tots els mapes tindran un *GameObject* anomenat “MapManager<Identificador del mapa>” amb el seu respectiu *script*. Aquest ens ajuda a controlar els esdeveniments de cadascun dels mapes. Els mapes també disposaran de *checkPoints*, que són punts del mapa on reapareix el personatge quan es mor. Aquests *checkPoints* són de gran utilitat, ja que eviten que el jugador hagi de recórrer un nombre excessiu de vegades algunes zones per on ja ha passat prèviament. Per tal d’activar esdeveniments del mapa com poden ser els *checkPoints*, utilitzarem zones invisibles (Un *GameObject* amb un *Trigger*) col·locades en punts escollits estratègicament en el mapa. Així doncs, en traspasar-les, el jugador els activarà automàticament sense necessitat de fer-ho a manualment.

2.6.1 Mapa 1

Aquest mapa serà el primer que visitarem. Té lloc sota el clavegueram d’una gran ciutat, i contindrà una sèrie de proves i diàlegs per a introduir al jugador en les mecàniques i la història del videojoc (Figura 24).

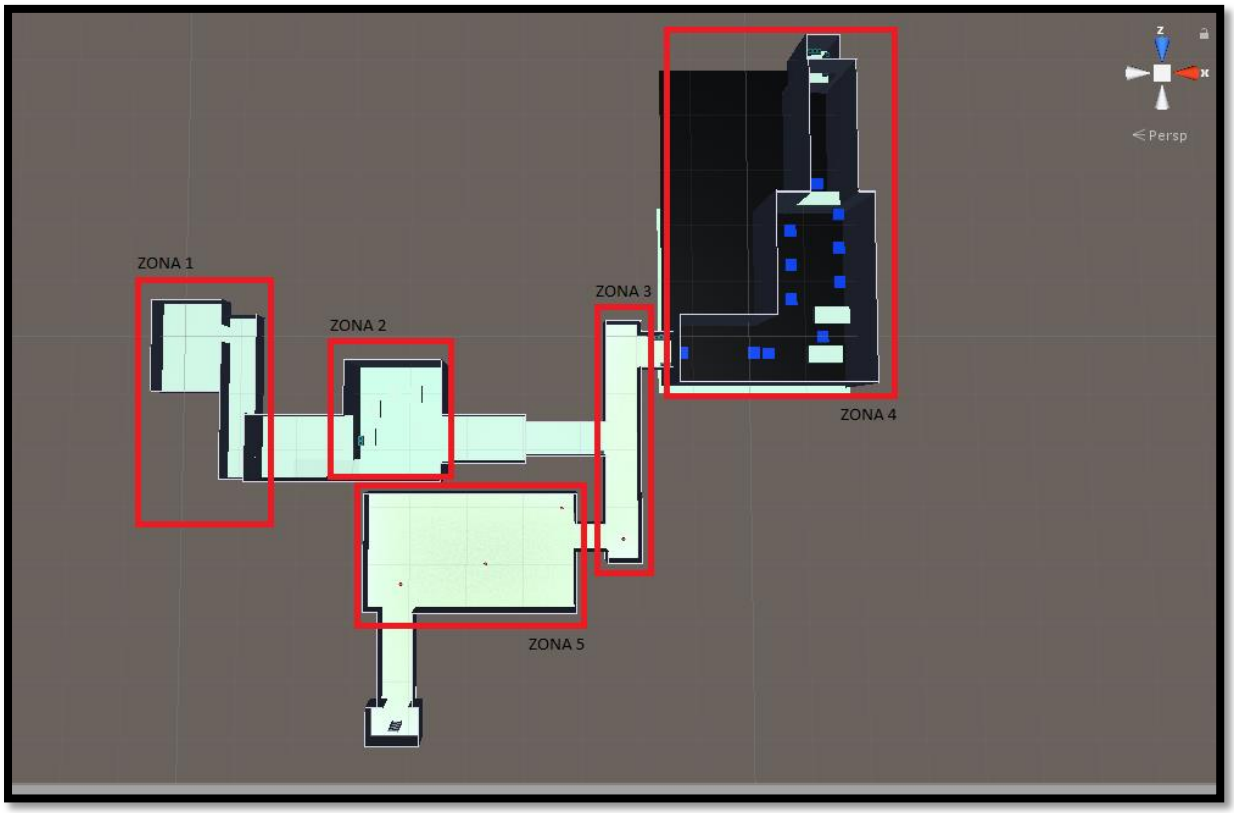

Figura 24: Perspectiva aèria del Mapa 1.

El Mapa 1 es divideix en 4 zones, que es descriuen a continuació:

- Zona 1: On explicarem com es mou el personatge, com utilitzar la càmera i a interactuar amb els objectes del mapa. Tan aviat el personatge es trobi a la Zona 1, apareixerà un quadre de diàleg on s'explicarà com moure's, com utilitzar la càmera i com interactuar amb els objectes del mapa.
- Zona 2: En aquesta segona zona, un cop el jugador ja ha adquirit els coneixements més bàsics, se li explicarà el salt i el doble salt. Contindrà varies plataformes de baixa dificultat.
- Zona 3: Aquí, es trobarà, per primer cop, amb un enemic, però encara no el podrà vèncer, ja que , en aquest moment, encara no ha adquirit cap arma.
- Zona 4: Superades unes plataformes mòbils, el personatge trobarà la seva primera arma, el tub. Es farà una breu introducció de les caixes de cristalls i dels cristalls com a moneda del joc.
- Zona 5: Després de superar l'enemic de la Zona 4, trobarà una gran sala on hi hauran diversos enemics, el que li permetrà posar a prova l'arma cos a cos i les seves combinacions d'atacs.

2.6.2 Mapa 2

Aquest mapa és el que es trobarà a continuació de les clavegueres, i on s'introduirà el funcionament de les armes. El protagonista apareixerà als canals de la gran ciutat, on ens sorprendran múltiples enemics, els quals s'hauran de combatre per a poder avançar (Figura 25).

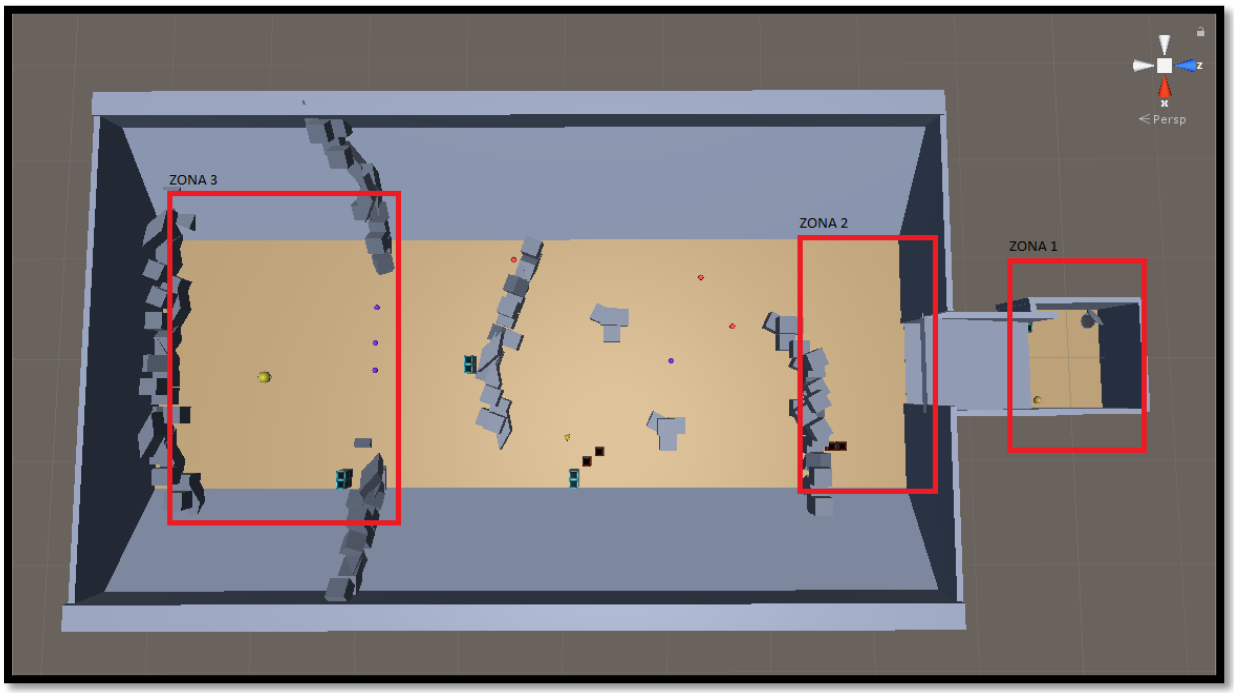

Figura 25: Perspectiva aèria del Mapa 2.

Com en el cas anterior, aquest segon mapa també el dividirem en zones:

- Zona 1: En sortir del clavegueram, es farà la introducció a la botiga, on hi haurà una barrera invisible que ens impedirà el pas fins que no desbloquegem una arma a la botiga.
- Zona 2: Aquesta zona es dedicarà a una petita introducció a les caixes de munició i també s'explicaran característiques dels combats amb armes a distància, com poden ser el bloqueig de rotació del personatge o l'ús de l'auto apuntat.
- Zona 3: Finalment, s'ensopegarà amb una zona infestada d'enemics i el seu líder al final del nivell.

2.7 Objectes

2.7.1 Objectes interactuables i NPC

Aquests objectes constaran principalment del model de l'objecte i una zona al seu voltant (un *Trigger*), que en ser penetrada pel jugador, activarà la possibilitat d'interactuar amb la tecla "E". Quan s'activi aquest funcionament, apareixerà un cartell que fins al moment romania invisible, en el que s'hi mostrarà un missatge per ajudar al jugador a conèixer la naturalesa de l'objecte, és a dir, que és interactuable. Aquest missatge gira de tal manera que sempre està orientat en direcció al personatge. Així ens assegurem que sempre podrà llegir-lo.

En el cas dels objectes recol·lectables, tant els cristalls com els paquets de munició comparteixen la mateixa estructura de components. Ambdós disposen de dos *Colliders,* un de més petit situat a l'interior que impedirà que el cristall traspassi els objectes (incluint-hi el terra), i un segon que tindrà la variant *Trigger* activada, ja que serà l'encarregat de detectar les col·lisions amb el jugador. En aquest cas, s'ha utilitzat un *Trigger* perquè no volem que aquest col·lisioni amb el jugador en el moment de recolectar-los.

2.7.2 Cristalls

Els cristalls representaran la moneda de canvi del nostre videojoc. Aquests, es poden trobar de dues formes: al destruir caixes de carregament de cristalls que el jugador anirà trobant al llarg del joc i al véncer enemics. Aquestes monedes es poden intercanviar per noves armes i munició a la botiga. Aquest *GameObject* disposarà d'un sol *script*, "Crystal.cs", que espera la col·lisió amb el jugador per sumar-li un cristall i, posteriorment, destruir-se (Figura 26).

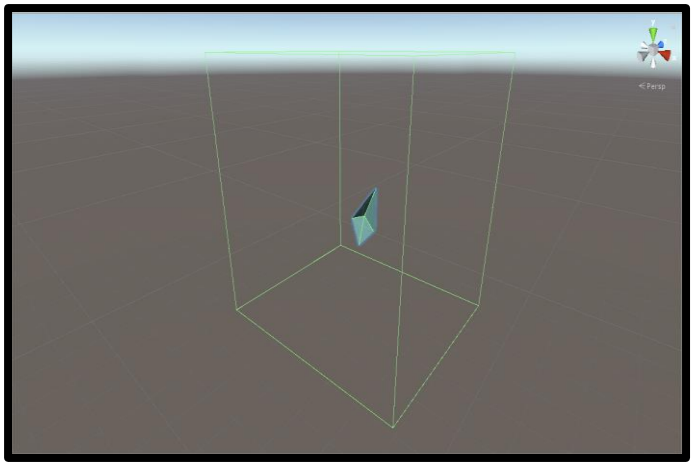

Figura 26: Model d'un cristall.

2.7.3 Paquet de munició

Tots els paquets de munició, independentment de l'arma a la que pertanyen, utilitzen el mateix *script*: "AMMOPack.cs". Aquest *script* té tres inputs: la id de l'arma a la que pertany, i el mínim i el màxim de munició que pot deixar anar. En el moment que col·lisioni amb el jugador, comprovarà la seva munició i, si no la té al màxim, generarà un número aleatori de munició (dins del rang assignat a l'input del *script*), afegirà a la munició al jugador i, després, es destruirà.

2.7.4 Caixa de Cristalls

Les caixes de cristalls consten de dos *scripts*. Un d'ells, l'anomenat "box.cs", aplicarà un força per simular la gravetat, i l'altre, que rep el nom de "CrystalBoxStatManager.cs", extendrà de la classe "StatManager.cs". Aquesta classe modifica la funció "die()" afegint-hi una explosió de cristalls en el moment de destruir-se (Figura 27).

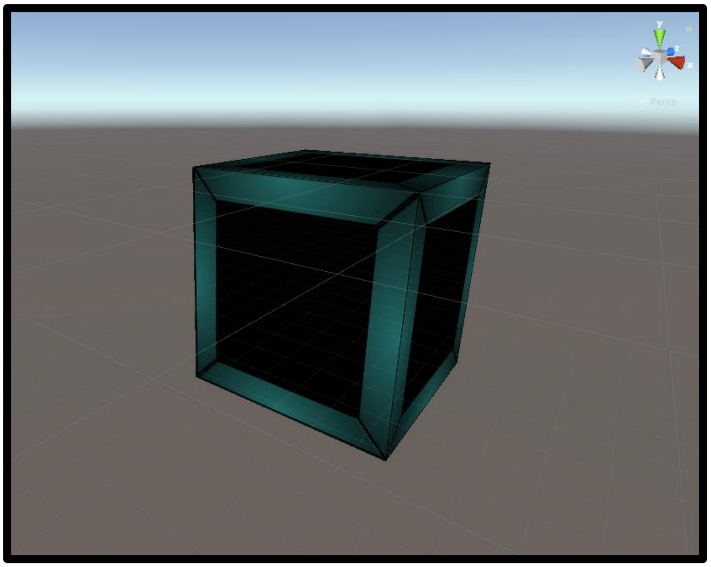

Figura 27: Model Caixa de cristalls.

2.7.5 Caixa Munició

La caixa de munició funciona pràcticament igual que la caixa de cristalls, però en el moment de trencar-se, deixa anar paquets de munició d'una arma seleccionada a l'atzar. Abans de soltar-los, això sí, ha de saber quins seran útils per al jugador. Per tal de saber-ho, li pregunta al "WeaponManager.cs" i aquest li retorna una llista de les armes que té en propietat i que utilitzen munició. De forma aleatòria es pot donar el cas que en la mateixa explosió caiguin dos paquets de munició de la mateixa caixa (Figura 28).

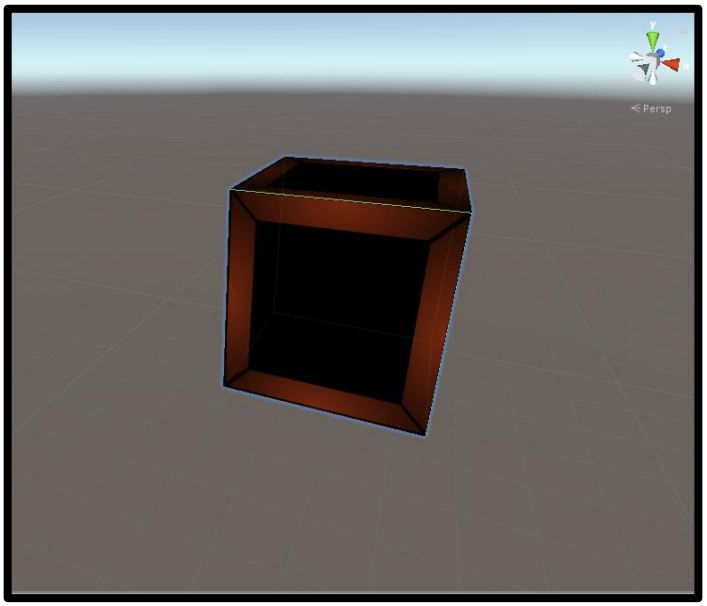

Figura 28: Model de Caixa de munició.

2.7.6 Plataformes

Les plataformes són objectes que tenen la funció d'ajudar al personatge a arribar a objectius inaccessibles, superant grans depressions del terreny. Les plataformes més bàsiques són, simplement, un objecte flotant on el jugador pot saltar-hi a sobre. Per a fer-ho més divertit, he creat unes plataformes que es mouen automàticament i de forma continuada.

Les plataformes automàtiques seran controlades pel *script* "Plarform.cs". Per cada una de les coordenades (X, Y, Z), el *script* disposarà dels següents inputs: distància a recórrer, direcció a la que es dirigeix i velocitat a la que es mou. Així doncs, tot i que gairebé totes les plataformes es mouran en direccions i velocitats diferents, es pot reutilitzar el mateix *script* en tots els casos, modificant únicament els inputs.

Capítol 3. Modelatge del personatge

Per crear el model del personatge s'ha utilitzat una tècnica que es basa en modelar a partir de dues imatges de referència, la frontal i la lateral (Figura 29). Des de la vista ortogonal del programa Blender col·locarem una de les imatges de fons en la perspectiva frontal i l'altra en una de les perspectives laterals.

A Partir d'aquestes imatges que utilitzarem de guia he creat un cub al centre i l'he anat modificant seguint la guia que ens ofereixen les dues imatges de referència i així fins a arribar a la figura en 3D desitjada.

En el moment de modelar, només modelarem la meitat del personatge, ja que, d'aquesta manera, podrem aplicar un *Modifier* (eina de Blender) de Blender anomenada "Mirror", que construirà l'altra meitat del personatge de forma automàtica i idèntica a la primera, i així ens assegurem que el model sigui totalment simètric.

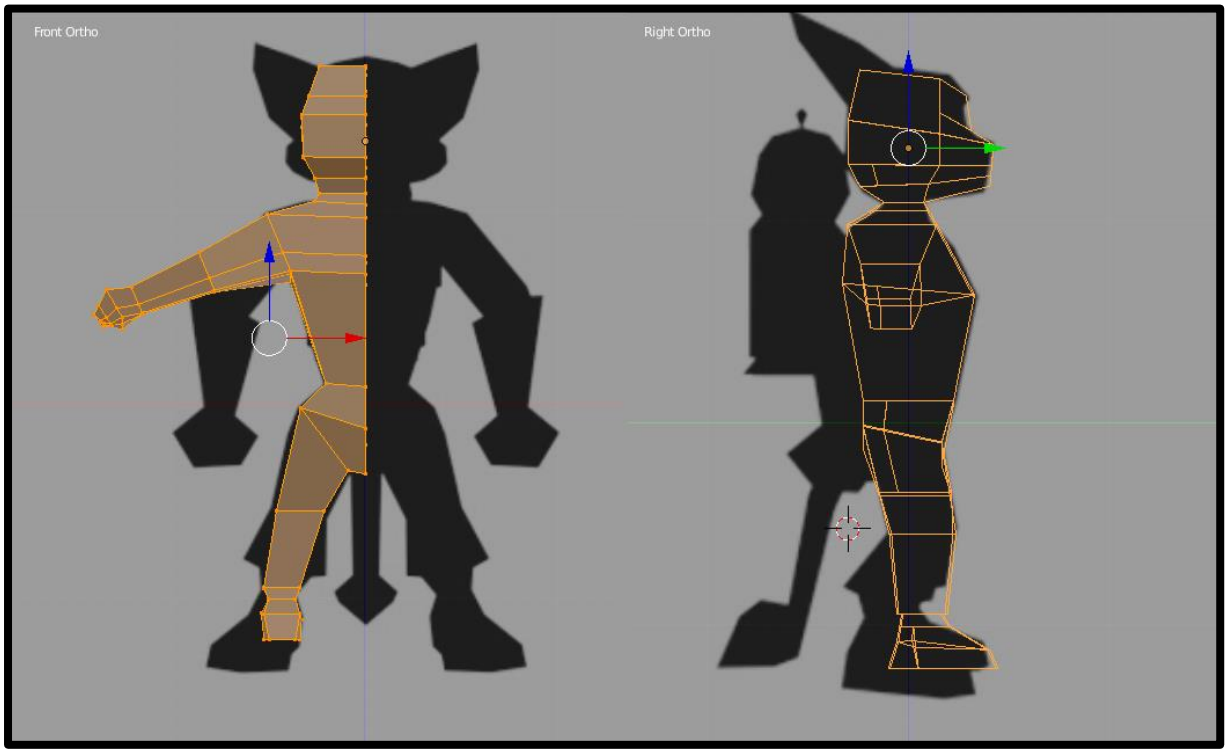

Figura 29: Tècnica utilitzada per a crear figures 3D.

3.1 *Rigging*

El *rigging* simularà els ossos del personatge (Figura 30) i serà imprescindible per a poder-lo moure després amb les animacions. A l'hora d'afegir el *rigging*, s'ha de tenir en compte el mapa de temperatura, ja que aquesta indica quins dels vèrtex del model es mouen quan es

mouen els nodes del *rig*. El *rig* és essencial per a fer animacions, ja que fa que el model es pugui moure amb facilitat i rapidesa per aconseguir la posició desitjada.

Per al nostre personatge s'ha creat un sistema de rigging centralitzat. És a dir, tots els nodes son fills de l'anterior fins a arribar al node central (en el nostre cas, el node situat a la cintura). Aquest sistema, com es pot apreciar a la Figura 30, és molt semblant a l'estructura òssia humana, encara que, òbviament, de forma simplificada i afegint-hi un node al cap. El fet que sigui semblant a la humana ens facilita la posterior creació de les animacions bàsiques, ja que disposem de moltes referències per guiar-nos.

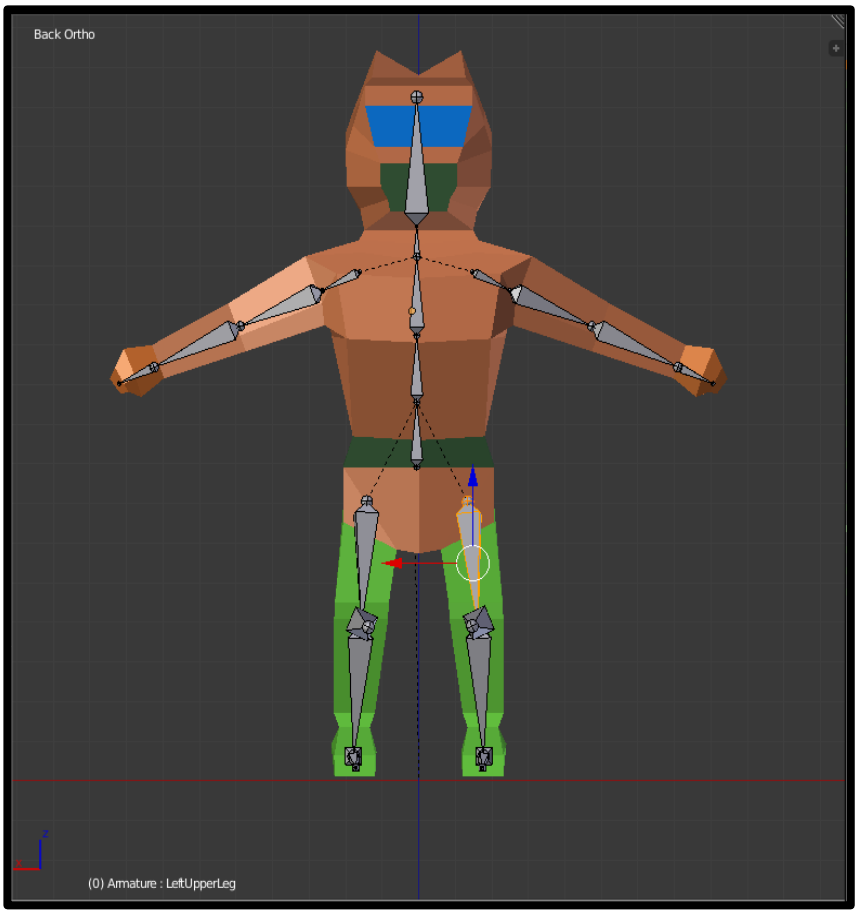

Figura 30: Sistema de *rigg* del personatge principal dins de Blender.

3.2 Animació

Les animacions són un conjunt de fotogrames amb una posició específica del model. Utilitzant el sistema *rigging* esmentat a l'apartat anterior, he creat una gran varietat d'animacions, concretament 30. Dins d'aquestes es troben animacions de caminar, correr (Figura 31), saltar, levitar i totes aquelles pertinents per a fer la combinació d'atacs amb l'arma cos a cos. Com es pot veure, no s'han enumerat les 30 animacions, ja que moltes

d'aquestes són variacions de les principals. Per a cada variant (Per exemple: caminar amb o sense arma) s'ha creat una animació diferent.

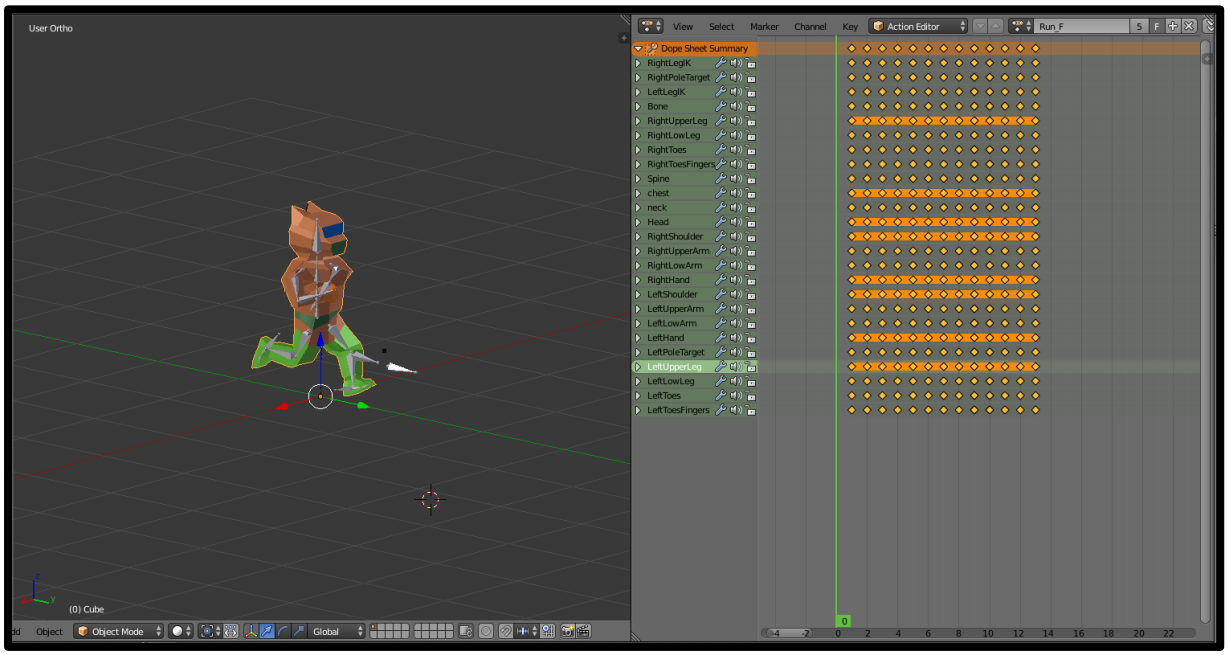

Figura 31: Estructura dels fotogrames d'una animació dins de Blender.

3.3 *Animator*

Finalment, per a poder controlar i decidir en cada moment quines animacions volem utilitzar dins el videojoc, Unity disposa d'un *animator* (una màquina d'estats). Aquest representa, mitjançant estats i transicions, l'animació on es troba i quina condició ens moura a la següent animació. Com es pot veure en la Figura 32, l'*animator* del personatge principal es divideix en dues zones, "With Weapon" (amb arma) i "Without Weapon" (sense arma). Dividir l'*animator* d'aquesta manera ens ajudar a controlar la variant més comú dins les nostres animacions, diferència entre portar o no l'arma equipada. D'aquesta manera, ens és molt més fàcil treballar de forma ordenada i controlada amb un gran nombre d'animacions i totes les seves variants.

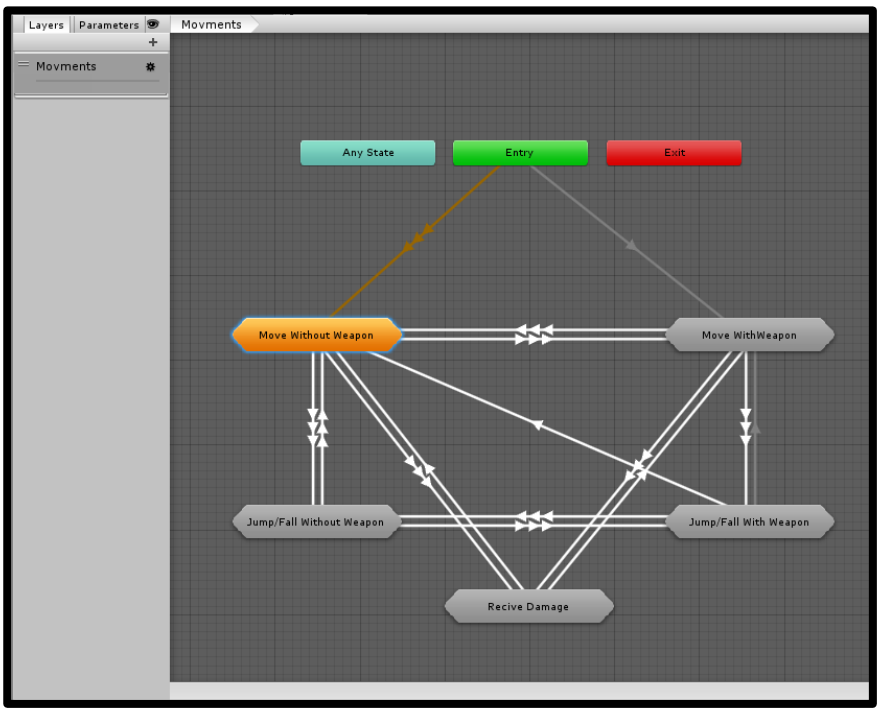

Figura 32: Arbre d'animacions del personatge principal.

Les transicions entre animacions es poden personalitzar perquè tinguin una transició més progressiva o bé sigui instantànies. També es pot habilitar perquè esperi al final de l'animació per fer la transició.

Per poder controlar les transicions de l'*animator*, aquest disposa de variables per a crear condicions a les transicions. Aquestes variables són accesibles desde qualsevol *script*. En la Figura 33, es pot veure que disposem d'una condició que verifica si la variable "grounded" és certa. Aquesta variable és modifica desde el *script* "MovementManager.cs" concretament a la funció "IsGrounded()".

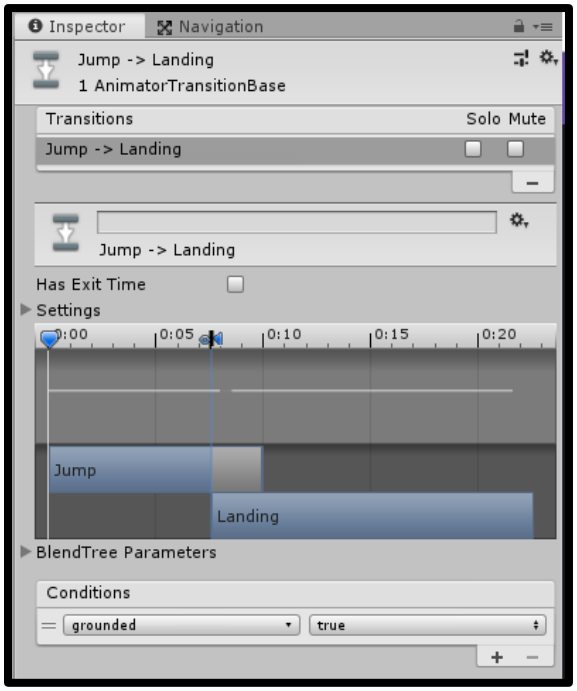

Figura 33: Transició entre l'animació "Jump" i l'animació "Landing".

Capítol 4. Execució del videojoc desenvolupat

El videojoc es troba dins un arxiu comprimit (Videojoc.rar). En descomprimir l'arxiu, extraurà vàries carpetes i un executable (CyberX.exe) amb una icona especial del videojoc. Fent doble clic sobre la icona del videojoc, aquest es podrà obrir i s'hi podrà jugar sense necessitat d'instal·lar cap programa. És important saber que actualment només es disposa d'una versió per a Windows.

Abans d'entrar dins el videojoc, s'obrirà una finestra on podrem configurar la qualitat dels gràfics i la resolució a la que volem executar el videojoc. Així doncs, es pot configurar perquè s'adigui al màxim a l'ordenador de cada usuari.

En obrir el joc el primer que ens trobarem serà el menú principal. Allí, si ja hem jugat prèviament, podem carregar alguna de les partides guardades anteriorment i, si no és així, en començarem una de nova.

Al començar-ne una de nova, apareixerem en el primer mapa, on ja estarà integrat el tutorial per aprendre a utilitzar el nostre personatge i un seguit d'obstacles que ens ajudaran a utilitzar i a entendre el funcionament del videojoc.

Capítol 5. Conclusió

Com ja se sospitava en un inici, fer aquest videojoc en tan sols uns mesos era un projecte molt ambiciós. Tot i això, els resultat aconseguits són molt positius, ja que s'han implementat gairebé totes les funcionalitats que es van proposar en un principi. Un cop acabat el projecte, agraeixo haver pres la decisió de definir un ordre de prioritats poc usual de tasques (comparat amb un desenvolupament ordinari de videojocs). Començar per les parts que ens eren més desconegudes em va aportar una visió més crítica sobre el projecte i em va ajudar a organitzar les tasques d'una forma més realista. Tot i que, com he esmentat abans, s'ha implementat gairebé tot el que es va proposar en el document inicial, es podrien crear noves funcionalitats i millores en algunes de les implementacions ja existents. En l'àmbit de l'IPO, tot i que he pogut gaudir de l'ajuda d'uns 15 usuaris d'edats i *hobbies* molt diferents, m'hagués agradat fer un estudi més professional. Malgrat això, degut al poc temps del que disposava per a la realització del treball i la creació del videojoc, això no ha sigut possible. Tot i així, cal dir que els 15 usuaris que han pogut provar el videojoc, amb les opinions i suggeriments aportats, han influenciat de forma molt positiva en el desenvolupament del videojoc.

Alguns aspectes que han quedat pendents són:

- Augmentar la varietat d'enemics
- Augmentar la varietat d'armes
- Crear nous models 3D per decorar el mapa
- Millorar la UI, per fer-la més agradable visualment
- Afegir la possibilitat de canviar d'idioma dins el joc
- Afegir la possibilitat de jugar amb un comandament de consola.

Pel que fa als programes utilitzats, he adquirit un nivell suficient a l'hora de crear models i animacions amb Blender. Tanmateix, cal esmentar que aquest programa és immens i el percentatge que s'ha explorat es ínfim comparat amb tot el que pot oferir. De la mateixa manera, a Unity, tot i que tampoc s'ha utilitzat tot el potencial del programa, considero que he adquirit un bon nivell pel que fa al funcionament de l'eina.

Per acabar, considero que ha estat un bona decisió escollir aquest tipus de projecte per al treball de final de grau, ja que considero que és un projecte de gran interés i molt desafiant, degut a la gran quantitat de disciplines que s'hi veuen implicades. He invertit moltes hores en ell, però en cap moment han estat forçades. A més a més, ha estat un procés molt divertit i enriquidor, motiu pel qual continuaré treballant en aquest videojoc, amb l'objectiu de poder-lo tenir acabat algun dia.

Webgrafia

- Unity (2019). Documentació Unity. Recuperat de https://docs.unity3d.com/Manual/index.html
- Unity (2019). Fòrum Unity. Recuperat de https://answers.unity.com/index.html
- Microsoft (2019). Documentació c#. Recuperat de https://docs.microsoft.com/es-es/dotnet/csharp/
- Maria Jose (2019). Font de lletra utilitzada per al videojoc. Recuperat de https://www.canva.com/learn/futuristic-fonts/
- FreeSound (2019). Àudios utilitzats dins el videojoc. Recuperat de https://freesound.org
- Ratchet And Clank(2019). Imatges de fons utilitzades en la Figura 29. Recuperat de https://polycount.com
- Anttis instrumentals(2019). Cançons utilitzades dins el videojoc. Recuperat de https://www.soundclick.com/bands3/default.cfm?bandID=1277008&content=songs